819-RD-001-003

EOSDIS Core System Project

EOSDIS Core System (ECS) Application Programming Interface (API) Interface Definition Document (IDD) for the ECS Project

Preliminary

October 1996

Hughes Information Technology Systems
Upper Marlboro, Maryland

EOSDIS Core System (ECS) Application Programming Interface (API) Interface Definition Document (IDD) for the ECS Project

Preliminary

October 1996

Prepared Under Contract NAS5-60000

SUBMITTED BY

R. E. Clinard /s/ 10/31/96

Robert E. Clinard, ECS CCB Chairman Date
EOSDIS Core System Project

Hughes Information Technology Systems
Upper Marlboro, Maryland

819-RD-001-003

This page intentionally left blank.

Preface

This document is submitted as required by the ECS Contract and does not require Government approval.

This document is under the control of the ECS Configuration Management Office. Any questions or proposed changes should be addressed to:

Data Management Office
The ECS Project Office
Hughes Information Technology Systems
1616 McCormick Drive
Upper Marlboro, MD 20774-5372

This page intentionally left blank.

Abstract

This Interface Definition Document (IDD) defines and describes the Application Programming Interfaces (APIs) that external users can use to invoke any of a set of ECS functions, such as searching the science data server.

This document describes what an API is and explains the pre-conditions that must exist in order to successfully call the APIs. This document describes each of the functions that are available to a generic ECS user and explains how to invoke them in a step by step process. The relevant objects (APIs) and methods are given and code segments are included to show the details of invoking that function. A section is provided that gives some insight to the ECS system via the philosophy behind parts of the system. This section also provides some high level hints and rules of thumb for the programmer to keep in mind, as well as an example to flesh out the process.

This document contains information pertaining to ECS Release B. This document is preliminary; the final version (for Release B) will be delivered with other final ("as-built") documentation for Release B.

Keywords: API, interface, search, order, acquire, subscription, subscribe, ingest, statusing, submit, advertisement, ingest

This page intentionally left blank.

Change Information Page

List of Effective Pages	
Page Number	**Issue**
Title	Preliminary
iii through xiv	Preliminary
1-1 through 1-2	Preliminary
2-1 through 2-2	Preliminary
3-1 through 3-10	Preliminary
4-1 through 4-56	Preliminary
5-1 through 5-2	Preliminary
A-1 through A-2	Preliminary
B-1 through B-2	Preliminary
AB-1 through AB-2	Preliminary

Document History			
Document Number	**Status/Issue**	**Publication Date**	**CCR Number**
819-RD-001-001	Preliminary	March 1996	96-0177
819-RD-001-002	Preliminary	August 1996	96-0998
819-RD-001-003	Preliminary	October 1996	96-1211

This page intentionally left blank.

Contents

4. Interface Services

5. API Object Descriptions

Figures

Tables

This page intentionally left blank.

1. Introduction

1.1 Identification

This Interface Definition Document (IDD) is a Required Document specified in Change Order 1 to the Earth Observing System (EOS) Data and Information System (EOSDIS) Core System (ECS) Contract (NAS5-60000).

1.2 Scope

This IDD defines and describes the Application Programming Interfaces (APIs) that external users can use to invoke any of a set of ECS functions, such as searching the science data server. Unless otherwise stated, all sections of this document apply to Release B. The information in this IDD is preliminary, and will not be finalized until delivery of Release B is complete.

This document defines an API and explains the pre-conditions that must exist in order to successfully interface with ECS via the APIs. It describes each of the functions that are available to an external ECS user and explains how to invoke them in a step by step process. The relevant objects (APIs) and methods are given and code segments are included to show the details of calling that function. A section is provided that provides some insight to the ECS system via the philosophy behind parts of the system. This section also provides some high level hints and rules of thumb for the programmer to keep in mind, as well as an example to flesh out the process.

The intended user of an API is any external user of ECS who can satisfy the requirements in Section 3 and can write the code necessary to interface with the APIs. The base of users is not restricted to Distributed Active Archive Centers (DAACs) or Science Computing Facilities (SCFs). Therefore, the functionality described in this document must be general enough to cover this broad user base. DAAC-specific capabilities that require more extensive access to ECS private services are documented in the particular DAAC's Interface Control Document.

The Earth Science Data and Information System (ESDIS) Project has responsibility for the development and maintenance of this IDD. Any changes in the ECS system that affect this interface will be reflected in revisions to this document. This IDD will be approved under the signatures of the ESDIS Project Manager.

1.3 Purpose and Objectives

This document is written to aid external users who wish to provide value added capabilities to the ECS in understanding APIs and using them to interact with the ECS. This IDD has objectives to define what an API is within the ECS context, explain the prerequisites to interfacing with ECS via APIs, explain step by step how to use them, and provide some additional insight via hints and an example.

The purpose of issuing interim or preliminary versions of this document prior to the final, as-built version to be delivered with Release B is to assist the DAACs, SCFs, and others who intend to extend or interoperate with the ECS Release B system. The material in the preliminary version should be used by those planning such extensions to gain an understanding of the skills and information required to design and implement extensions, and to gain an appreciation of the scope of the work. While these public interfaces (see Section 3.2) are expected to be fairly static, they are likely to undergo some changes during Release B construction, ranging from method signature changes to replacement of entire objects. Users of this document should be aware of the nature of documentation at this stage of development; dependence on precise details is not advised and is at the user's risk. Note also that the authoritative source for documentation of these public APIs is the final version of the Release B CSMS/SDPS Internal Interface Control Document for the ECS Project (DID 313).

1.4 Status and Schedule

This version of the IDD is being prepared to assist parties interested in extending or interoperating with ECS Release B, and its content is relevant to Release B only. Its content is preliminary and will updated to provide additional format and content. It will be released in final form when Release B is delivered. This document will be submitted to the ESDIS CCB as a Configuration Control Board (CCB) document.

This IDD is a Required Document specified in Change Order 1 to Contract number NAS5-60000. Government approval is not required. This document will be configuration controlled via the Contractor CCB process. Changes may be submitted for consideration by Contractor CCB under the normal change process at any time.

Within this document are some interfaces that have associated TBRs, TBSs and/or TBDs. A table providing a Work-off Plan is in Appendix A. This plan provides the following information:

a. IDD Interface Issue Number
b. IDD Reference Paragraph
c. Issue Priority
d. IDD Issue Description
e. Work-off Plan
f. Projected Date of Issue Resolution

1.5 Organization

This document is organized in 5 sections plus appendices. Section 1 provides information regarding the identification, scope, purpose and objectives, and organization of this document. Section 2 contains information about documentation relevant to this IDD, including parent, applicable, and information documents. Section 3 provides an overview the prerequisites that must be accomplished in order to interface with ECS via the APIs. Section 4 provides an overview of the functional interfaces. It describes each functional interface and describes the process required to invoke that function. Section 5 contains a description of the APIs required to invoke the functions described in Section 4. A Work-off plan is presented in Appendix A. An acronym list is in Appendix B. Design philosophy, tips, and an example are presented in Appendix C.

2. Related Documentation

2.1 Parent Document

The following documents are the parents from which this document's scope and content are derived:

193-208-SE1-001	Methodology for Definition of External Interfaces for the ECS Project
301-CD-002-003	System Implementation Plan for the ECS Project
304-CD-002-002	Science Data Processing Segment (SDPS) Requirements Specification for the ECS Project
304-CD-003-002	Communications and System Management Segment (CSMS) Requirements Specification for the ECS Project
423-41-01	Goddard Space Flight Center, EOSDIS Core System (ECS) Statement of Work
423-41-02	Goddard Space Flight Center, Functional and Performance Requirements Specification for the EOSDIS Core System (ECS)

2.2 Applicable Documents

The following documents are referenced herein and are directly applicable to this document. In the event of conflict between any of these documents and this document, this document shall take precedence. Internet links cannot be guaranteed for accuracy or currency.

305-CD-020-002	Overview of Release B SDPS/CSMS System Design Specification for the ECS Project
305-CD-024-002	Release B SDPS Data Server Subsystem Design Specification for the ECS Project
305-CD-028-002	Release B CSMS Communication Subsystem Design Specification for the ECS Project
313-CD-006-002	Release B CSMS/SDPS Internal Interface Control Document for the ECS Project
194-WP-902-002	ECS Science Requirements Summary White Paper
505-10-23	Goddard Space Flight Center, EOSDIS Security Policy and Guidelines

2.3 Information Documents

The following documents, although not directly applicable, amplify or clarify the information presented in this document. These references are not binding on this document.

none	Object-Oriented Modeling and Design, James Rumbaugh, et al

3. Prerequisites to Using APIs

3.1 Overview

This section provides an explanation of the conditions that must be satisfied and the knowledge a programmer should have before a he can successfully have his application program call an API. This section will cover the following: the knowledge that is necessary in order to write the code which calls these capabilities, the ECS architecture, the key mechanisms within ECS, test requirements, required software libraries, the administrative path to follow to gain approval to use the ECS APIs for a particular application, and the process by which the new application program may be advertised for use by others.

This document assumes a context where the user wishes to create a unique capability not offered within the ECS system, for instance automating a daily search and retrieval of data products containing certain parameters. Within this context there are certain interactions with ECS which must be automated. It is for this purpose that APIs are useful -- they allow the user to interact with the system from within his own application.

3.2 API Within ECS Context

ECS was designed using an object-oriented design methodology (Rumbaugh), and has been constructed using an object-oriented language, C++, in addition to integrated COTS products. An API within the ECS context is an object that an external user may call in order to invoke an ECS service, such as searching the science data server or ordering data. As will be seen later in Section 4, a single API does not correspond to a single service. The external user may have to call several different APIs to invoke a particular service.

These APIs are the same APIs used within ECS, and consist of that group of objects that can be called by an object outside of the called object's particular computer software configuration item (CSCI). This type of object is called a public object. Conversely, some objects within ECS can only be called by other objects within their particular CSCI. These are called private objects.

The intended user of an API is any external user of ECS who can satisfy the requirements in Section 3 and can write the code necessary to call the APIs. The base of users is not restricted to Distributed Active Archive Centers (DAACs) or Science Computing Facilities (SCFs). Therefore, the functionality described in this document must be general enough to cover this broad user base. DAAC-specific capabilities that require more extensive access to ECS are documented in the particular DAAC's Interface Control Document.

3.3 Required Knowledge

The ECS APIs which will be called to invoke a particular service are coded in C++. The external programmer writing an application which calls these APIs must therefore be capable of writing C++ code to accomplish this task. The entire application, both science code and the ECS API interface, may be written in C++ if the user chooses. On the other hand, he may write or reuse science code in any language and provide a method for it to communicate with the ECS API interface code.

Also helpful in designing the code that interfaces with the ECS APIs is a knowledge of Object-Oriented Design (OOD). This is useful for two reasons. First the ECS architecture is object-oriented. Since the user must interact within this framework, an understanding of it is useful. Second, C++ is an object-oriented programming language. Since the user must program some portion of his application in this language an understanding of OOD gives the programmer more insight into the language.

In general, "object-oriented" means that the software is organized as a collection of discrete objects. Each object has both data structure and behavior. In contrast conventional programming languages have only loosely connected data structure and behavior. A good source for more information about this subject is *Object-Oriented Modeling and Design* by James Rumbaugh, et al.

3.4 ECS Architecture

The ECS design has been partitioned into three segments, namely:

- A Science Data Processing Segment (SDPS) which is responsible for ECS applicationswhich provide
  - data management and archiving functions,
  - a processing environment for the execution of science software,
  - external interfaces for the acquisition of data needed for processing or intended archiving, and– functions which support the search and retrieval of ECS managed data by science and other users.
- A Communications and System Management Segment (CSMS) which is responsible for all communications, networking, and enterprise management functions, including
  - a distributed applications and operating system infrastructure,
  - various communications services such as electronic mail and file transfer,
  - billing and accounting,
  - security,
  - monitoring and management of networking, system, and application resources,
  - access control, security, time synchronization, and reliable communications among local area network services and external network connectivity, and
- A Flight Operations Segment (FOS) which is responsible for space craft and instrument command and control functions.

This discussion addresses the Release B capabilities of Science Data Processing Segment and the Communications and System Management Segment. It is from the capabilities of these two segments that the APIs will be drawn.

The following sub-sections provide an overview of the SDPS and CSMS subsystems.

3.4.1 SDPS Segment Architecture

SDPS is composed of seven subsystems which provide the hardware and software resources needed to implement the SDPS functionality. This section provides a brief review of its capabilities.

The SDPS supports the services required to ingest, process, archive, access and manage science data and related information from the entire EOSDIS. Further details on these objectives are provided in ECS Science Requirements Summary White Paper (194-WP-902-002).

The subsystems can be grouped into the following four categories:

- Data storage and management is provided by the Data Server Subsystem (DSS), with the functions needed to archive science data, search for and retrieve archived data, manage the archives, and stage data resources needed as input to science software or resulting as output from their execution.
- Data search and retrieval is provided by the science user interface functions in the Client Subsystem (CLS), by data search support functions in the Data Management Subsystem (DMS), and by capabilities in the Interoperability Subsystem (IOS) which assist users in locating services and data of interest to them and their projects.
- Data processing is provided by the Data Processing Subsystem (DPS) for the science software; and by capabilities for long and short term planning of science data processing, as well as by management of the production environment provided by the Planning Subsystem (PLS).
- Data ingest is provided by the Ingest Subsystem (INS), which interfaces with external applications and provides data staging capabilities and storage for an approximately 1-year buffer of Level 0 data.

The following sub-sections provide brief overviews for each of these subsystems.

3.4.1.1 Client Subsystem (CLS)

The SDPS client subsystem has three main objectives:

- provide earth science users with an interface via which they can access ECS services and data
- offer an environment into which science users can integrate their own tools
- give science programs access to the ECS services, as well as direct access to ECS data

The client subsystem software, therefore, consists of graphic user interface (GUI) programs, tools for displaying the various kinds of ECS data (e.g., images, documents, tables), and libraries representing the client API of ECS services.

Modern user interfaces are based on an object paradigm. The SDPS client subsystem is no exception; the graphic user interface programs follow an object oriented design. The design is built around a core set of 'root' objects from which all other GUI software inherits its behavior. This leads to a consistent look and feel. This core set is called the desktop. The remainder of the software is collectively called the workbench.

For Release A, the client subsystem consists of the desktop, a user interface which allows users to search and browse a database describing the data and services available within ECS (the Advertising Service), and a data visualization tool (EOSView). The remainder of the Release A user interface is provided by an enhanced version of the V0 System Client (also referred to as the Release A Client). It provides data search and access for ECS science data, and a browsing interface for Guide documents and other types of ECS document data.

In Release B the V0 System Client is replaced, but the existing V0 client is supported through the V0 interoperability gateway. A new set of client tools is provided to provide search and access of ECS science data sets. The data search tool is separated from the product request tool in order to allow users more flexibility in the way they acquire data. Users will no longer be required to submit a search for data before ordering. If the user knows the location of data, an order can be constructed without the search phase. These tools along with other new tools are collectively called the "workbench".

3.4.1.2 Interoperability Subsystem (IOS)

The SDPS is architected as a collection of distributed applications. They need support by a distributed operating system and communications services. These are part of the CSMS and are described in the CSS Design Specification [305-CD-028-001]. To these functions, the SDPS interoperability subsystem adds an "advertising service." It maintains a database of information about the services and data offered by ECS, and provides interfaces for searching this database and for browsing through related information items. The Client Subsystem provides a user interface which enables scientists to locate services and data that may be of interest to them.

The full functionality of the advertising service is implemented in Release A. The only enhancements that will be made to the advertising service in Release B are to incorporate the Earth Science Query Language interface and the standard query protocol.

3.4.1.3 Data Management Subsystem (DMS)

The Data Management subsystem provides three main functions:

- Provide a dispersed community of science users with services to search and access data from a set of data repositories (however, the repositories themselves and their search, access, and data management functions are part of the Data Server subsystem).
- Allow those scientists to obtain descriptions for the data offered by these repositories. This also includes descriptions of attributes about the data and the valid values for those attributes.
- Provide data search and access gateways between ECS and external information systems.

The subsystem includes distributed search and retrieval functions and corresponding site interfaces. Release A uses the capabilities of the Version 0 IMS to provide both the user search interface, and the intersite search functions. In Release B, the Data Management subsystem provides the distributed search and access capabilities across wide area and local area networks using ECS query languages and protocols. There is a common language and protocol across this subsystem, Interoperability subsystem, and the Data Server subsystem.

3.4.1.4 Data Server Subsystem (DSS)

The subsystem provides the physical storage access and management functions for the ECS earth science data repositories. Other subsystems can access it directly or via the data management subsystem (if they need assistance with searches across several of these repositories). The subsystem also includes the capabilities needed to distribute bulk data via electronic file transfer or physical media. The main components of the subsystem are the following:

- Database Management System - SDPS will use an off-the-shelf DBMS (Illustra) to manage its earth science data and implement spatial searching, as well as for the more traditional types of data (e.g., system administrative and operational data). It will use a document management system to provide storage and information retrieval for guide documents, scientific articles, and other types of document data.
- File Storage Management Systems - they are used to provide archival and staging storage for large volumes of data.
- Data Type Libraries - they are an example of dynamic linked libraries (DLLs) and they will implement unique functionality for earth science and related data (e.g., spatial search algorithms and translations among coordinate systems). The libraries will interface with the data storage facilities, i.e., the database and file storage management systems.

The type library concept is at the heart of the DSS, and is the key to achieving the long term goal of providing database management capabilities for earth science data. In analogy to a database management system, each data type is registered in the data server schema, which describes the capabilities provided by its type library. The type library will then be accessed via an API which will provide object-relational database access capabilities.

3.4.1.5 Ingest Subsystem (INS)

The subsystem deals with the initial reception of all data received at an EOSDIS facility and triggers subsequent archiving and processing of the data.

Given the variety of possible data formats and structures, each external interface, and each ad-hoc ingest task may have unique aspects. Therefore, the ingest subsystem is organized into a collection of software components (e.g., ingest management software, translation tools, media handling software) from which those required in a specific situation can be readily configured. The resultant configuration is called an ingest client. Ingest clients can operate on a continuous basis to serve a routine external interface; or they may exist only for the duration of a specific ad-hoc ingest task.

The ingest subsystem also standardizes on a number of possible application protocols for negotiating an ingest operation, either in response to an external notification, or by polling known data locations for requests and data. The subsystem will use the components of the general ECS external interface architecture.

3.4.1.6 Data Processing Subsystem (DPS)

The main components of the data processing subsystem - the science algorithms - are provided by the science teams. The data processing subsystem provides the necessary hardware resources, as well as a software environment for queuing, dispatching and managing the execution of these algorithms. The processing environment is highly distributed and consists of heterogeneous computing platforms.

The DPS also interacts with the DSS to cause the staging and de-staging of data resources in synchronization with processing requirements.

3.4.1.7 Planning Subsystem (PLS)

The Planning Subsystem provides the functions needed to plan routine data processing, schedule on-demand processing, and dispatch and manage processing requests. The subsystem provides access to the data production schedules at each site, and provides management functions for handling deviations from the schedule to operations and science users.

3.4.2 CSMS Segment Architecture

Release B has been designed as a fully distributed, heterogeneous system. To implement this, SDPS will make use of the services which are provided by Communications and System Management Segment. An overview of this segment can be seen in Fig. 3-2.

SDPS relies extensively on the security management and authentication services provided by CSMS, and will add security services in areas where SDPS software components need to provide security for internally managed objects whose structure is transparent to CSMS.

The SDPS design makes use of a universal method for referencing persistent objects, called the Universal Reference (UR). URs encapsulate the identifier of a data object, as well as a (location independent) network name of an SDPS service which can interpret the object identifier and access the object. The CSMS Directory/Naming Service will be responsible for providing and maintaining the network names of SDPS services.

The CSMS accomplishes the interconnection of users and service providers, transfer of information between ECS (and many EOSDIS) components, and enterprise management of all ECS components. It supports and interacts with the Science Data Processing Segment (SDPS) and the Flight Operations Segment (FOS).

The services provided by CSMS at the System Monitoring and Coordination Center, (SMC) located at Goddard Space Flight Center (GSFC), are collectively referred to as Enterprise Monitoring and Coordination (EMC) throughout this document. In the same context, services provided by CSMS at Distributed Active Archive Centers (DAACs) and the EOC (sites) are collectively referred to as Local System Management (LSM).

At its highest design level, CSMS consists of three parts:

- System Management Subsystem (MSS) which is a collection of applications which manage all ECS resources, including all SDPS, FOS, ISS, and CSS components. MSS directly uses CSS services.
- Communications Subsystem (CSS) is a collection of services providing flexible interoperability and information transfer between clients and servers. CSS services correspond loosely to layers 5-7 of the Open Systems Interconnection Reference Model (OSI-RM).
- Internetworking Subsystem (ISS) is a layered stack of communications services corresponding to layers 1-4 of the OSI-RM. CSS services reside over, and employ, ISS services.

The following sub-sections briefly describe the CSMS subsystems and characterize their relationships with one another, SDPS and FOS, and external entities discussed above. More detailed material is provided in the corresponding CDRL 305 subsystem design documents.

3.4.2.1 Communications Subsystem (CSS)

CSS plays a key role in the interoperation of the SDPS subsystems. SDPS applications follow an object-oriented design. That is, their lowest level software components are "software objects". SDPS also implements a distributed design, in other words, its software objects are distributed across many platforms. For the software objects to communicate with each other requires a "distributed object" communications environment. This environment is provided by CSS, using off-the-shelf technology (OO-DCE from Hewlett-Packard) augmented with some custom software. The environment allows software objects to communicate with each other reliably, synchronously as well as asynchronously, via interfaces that make the location of a software object and the specifics of the communications mechanisms transparent to the application.

In addition, CSS provides the infrastructural services for the distributed object environment. They are based on the Distributed Computing Environment (DCE) from the Open Software Foundation (OSF). DCE includes a number of basic services needed to develop distributed applications, such as remote procedure calls (rpc), distributed file services (DFS), directory and naming services, security services, and time services.

Finally, CSS provides a set of common facilities, which include legacy communications services required within the ECS infrastructure and at the external interfaces for file transfer, electronic mail, bulletin board and remote terminal support. The Object Services support all ECS applications with interprocess communication and specialized infrastructural services such as security, directory, time, asynchronous message passing, event logging, lifecycle service, transaction processing and World Wide Web (WWW) service. The Distributed Object Framework is a collection of a set of core object services, collectively providing object-oriented client server development and interaction amongst applications.

3.4.2.2 Management Subsystem (MSS)

The Management Subsystem (MSS) provides enterprise management (network and system management) for all ECS resources: commercial hardware (including computers, peripherals, and network routing devices), commercial software, and custom software. With few exceptions, the management services will be fully decentralized, so that no single point of failure exists which would preclude the system from continuing to operate or system operations and management to come to a halt.

MSS provides two levels of an ECS management view: the local (site/DAAC specific) view, provided by Local System Management (LSM), and the enterprise view, provided by the Enterprise Monitoring and Coordination (EMC) at the SMC.

Enterprise management relies on the collection of information about the managed resources, and the ability to send notifications to those resources. For network devices, computing platforms, and some commercial of the shelf software, MSS relies on software called "agents" which are usually located on the same device/platform and interact with the device's or platform's control and application software, or the commercial software product.

However, a large portion of the ECS applications software is custom developed, and some of this software - the science software - is externally supplied. For these components, MSS provides a set of interfaces via which these components can provide information to MSS (e.g., about events which are of interest to system management such as the receipt of a user request or the detection of a software failure). These interfaces also allow applications to accept commands from MSS, provided to MSS from M&O consoles (e.g., an instruction to shut down a particular component).

Applications which do not interact with MSS directly will be monitored by software which acts as their "proxies". For example, the Data Processing Subsystem (DPS) acts as the proxy for the science software it executes. It notifies MSS of events such as the dispatching or completion of a PGE, or its abnormal termination.

ECS selected HP OpenView as the centerpiece of its system management solution, and is augmenting it with other commercially available "agents", as well as custom developed software (e.g., the applications interfaces mentioned above). The information collected via the MSS interfaces from the various ECS resources is consolidated into an event history database on a regular basis (every 15-to 30 minutes) as well as on demand, when necessitated by an operator inquiry. The database is managed by Sybase, and Sybase query and report writing capabilities will be used to extract regular and ad-hoc reports from it. Extracts and summaries of this information will be further consolidated on a system wide basis by forwarding it to the SMC.

MSS provides fault and performance management and other general system management functions such as security management (providing administration of identifications, passwords, and profiles); configuration management for ECS software, hardware, and documents; billing and accounting; report generation; trending; and mode management (operational, test, simulation, etc.)

3.4.2.3 Internetworking Subsystem (ISS)

The ISS is a layered stack of communications services corresponding to layers 1-4 of the Open Systems Interconnect Reference Model (OSI-RM). The ISS provides local area networking (LAN) services at ECS installations to interconnect and transport data among ECS resources. The ISS includes all components associated with LAN services including routing, switching, and cabling as well as network interface units and communications protocols within ECS resources.

The ISS also provides access services to link the ECS LAN services to Government-furnished wide-area networks (WANs), point-to-point links and institutional network services. Examples include the NASA Science Internet (NSI), Program Support Communications Network (PSCN), and various campus networks "adjoining" ECS installations. More detail of ISS is provided in Section 5 of this document.

3.5 Key Mechanisms

In the context of object technology, the design of a class embodies the knowledge of how individual classes behave. A key mechanism is a design decision about how collections of objects cooperate. In this sense, a key mechanism is a design pattern aimed at solving a recurring problem.

A full explanation of the key mechanisms is found in Chapter 6 of DID 313 Release B CSMS/SDPS Internal Interface Control Document.

3.5.1 Distributed Computing Environment (DCE) Requirements

The Distributed Computing Environment (DCE) Core Services act as a layer between the DCE Distributed Application and the platform Operating System, as illustrated in Figure 3.5-1, “DCE Services Architecture.” These Core Services include DCE Threads, DCE RPC, DCE Directory Services, DCE Distributed Time Services, and DCE Security Services. Additional extended services include Distributed File Services and Diskless Support Services. The OODCE block represents a third party extension selected for use within ECS. [1]

The DCE services provide an advanced distributed environment supporting distributed file services, directory and security services that are transparent to the user. These DCE services also provide an advanced foundation for sophisticated distributed application development.

A user who wishes to interface with ECS via the APIs must be within a known DCE cell. The version of DCE required is OSF DCE 1.1. A user may learn how to register his DCE cell that is external to ECS by contacting the EOSDIS Security Office at GSFC, specifically the ESDIS Information Technical Security Official in Code 505. This is documented in the EOSDIS Security Policy and Guidelines (505-10-23).

[1] See section 3.2 “ Advanced DCE -based Development Environments” for details

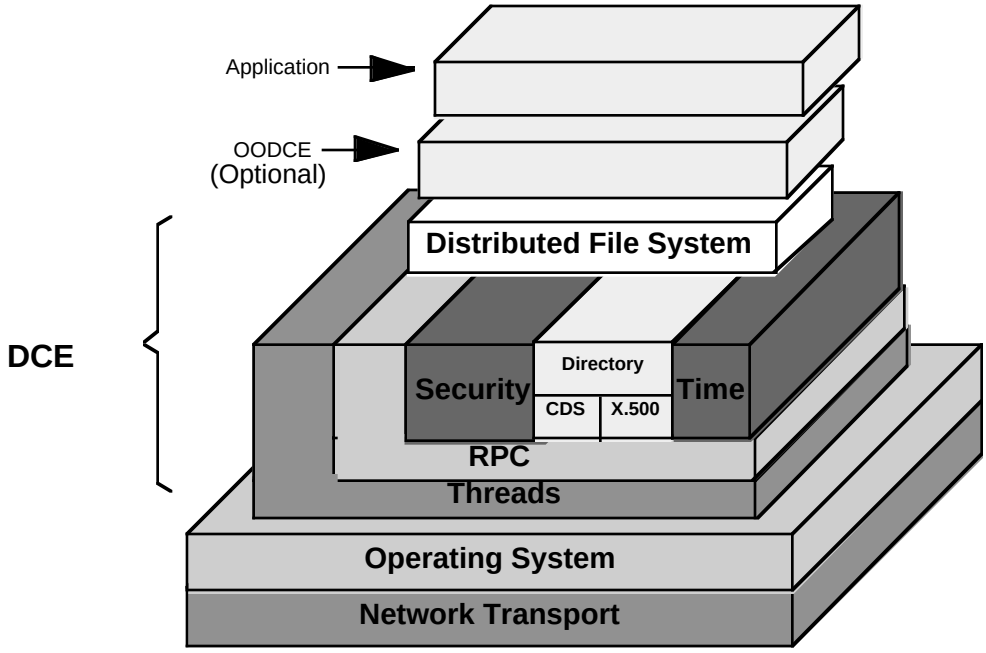

Figure 3.5-1. DCE Services Architecture

3.5.2 Security Requirements

Security issues are twofold -- electronic security for the distributed computing system and operational security.

DCE provides the electronic security services for ECS. Operational security issues are handled during the interface negotiation process in section 3.8.

3.6 Required Software Libraries

The primary development platform for ECS is the SUN; however, there are several other platforms that are also supported. Table 3.6-1 lists the general platforms used within ECS with the appropriate baseline compilers and operating system versions for Release B.

Table 3.6-1. Supported Platforms and Associated Software

Platform	Operating System	Compiler(s)	Version
SUN	Solaris 2.5.1	SPARCompiler C++	4.0
HP	HPUX 10.01	C++	10.01
SGI	IRIX 6.2	MIPSPro C++ KAI C++ (third party)	7.0 3.06
DEC	Digital UNIX 4	C++	5.4
IBM	AIX 4.1	C++	3.1.3.0

The following vendor software libraries are required for a user to access ECS via the APIs. Table 3.6-2 lists these products

Table 3.6-2. Required Software

Function	Product Name	Version	Comment
C++ Libraries	RogueWave Tools.h++	7.0.2	
C++ Libraries	Roguewave DBTools.h++	2.0	
DCE Tool	OSF DCE App Dev Kit	1.1	Not available for IBM*
DCE Tool	OSF DCE Client	1.1	Not available for IBM*
DCE Tool	HP OODCE for OSF DCE 1.1	n/a	ECS Performed custom port for SUN, SGI, DEC and IBM**

* It is not possible to access ECS via the ECS APIs using the IBM platform at this time

** For availability contact: ECS Contracts Department, Hughes Information Technology Systems, 1616 McCormick Ave., Upper Marlboro, MD, 20774

The COTS products listed in Table 3.6-3 are not required to interface with ECS via the ECS APIs. These products are useful in support of developing the software used to interface with the APIs.

Table 3.6-3. API Support Products

Function	Product Name	Version	Comment
DCE Tool	DCE Cell Dir. Service	1.1	
DCE Tool	DCE DFS	1.1	
DCE Tool	DCE Sec. Server	1.1	
DCE Tool	DFS Server	n/a	
Internet Browser	Netscape Browser	2.02	
RDBMS	Sybase Open Client	11	

The COTS and libraries in Table 3.6-4 are not required to interface with ECS via the ECS APIs. These products are necessary should the user wish to replicate the ECS environment.

Table 3.6-4. Other Products

Function	Product Name	Version	Comment
Archive	AMASS	n/a	
Billing&Accounting	SmartStream	4.0	
Billing&Accounting	SmartStream Server	4.0	
Code Analysis Tool	DDTS	n/a	
Compiler	Perl	5.003	
Compiler	Accugraph PNM	6.0.950329/2.1	
Config. Mgt. Tool	ClearCase Client	2.1	
GUI Tool	BuilderXcessory	3.5.1	
GUI Tool	Epak	2.5	
Integrated Logistic Mgt.	UNIFY/ACCELL	IDS2.0.7.2.0 dev	
Integrated Logistic Mgt.	XRP-II Dev. Version	3.0	
Internet Browser	Netscape Browser	2.02	
Internet Server	Netscape Enterprise Server	2.02	
Network Tool	HP OpenView	4.1	
Network Tool	HP OpenView NNM Dev. Kit	4.1	
Network Tool	SNMP Agent (Peer Networks)	2.2	
Network Tool	Tivoli Client	3.0	
Network Tool	Tivoli Mgt. Platform	3.0.1	
Network Tool	Tivoli Plus	2.5	
Network Tool	Tivoli/Admin	3.0	
Network Tool	Tivoli/AEF	3.0	
Network Tool	Tivoli/Clients	3.0, 2.5	3.0 for HP, Sun; 2.5 for SGI
Network Tool	Tivoli/Courier	3.0	
Network Tool	Tivoli/EIF (requires TEC)	2.6	
Network Tool	Tivoli/Enterprise Console	2.6	
Network Tool	Tivoli/Sentry	3.0	
Network Tool	Unix Mail Server	n/a	
Office Automation	Zmail	3.2.0	
OODBMS Tool	Illustra Iibmi/C++	3.2	
OODBMS Tool	Illustra Server	3.2	
Performance Tool	Autosys	3.3	
Performance Tool	Autosys Expert	3.3	
Performance Tool	Autosys Remote Agent	3.3	
RDBMS	Sybase Open Client	11	
RDBMS	Sybase SQL Server	11.01	
RDBMS	Sybase SQR Workbench	3.0.5	
RDBMS	Sybase SQS	2.2.1	
RDBMS	Sybase Xaclient	n/a	
Report Generator Tool	IQ	5.1.00	
Search Engine	Topic Server	1.0.2	
Search Engine	VDK	1.0.3	
System Monitoring Tool	ARWeb	1.1	
System Monitoring Tool	Remedy (ARS)	2.02	
Visualization Tool	IDL	4.0	

Updated information, based on prototyping of ECS extensions and component reuse, may be found on the World Wide Web at TBD-17.

3.7 Testing

User software testing issues will be handled during the interface negotiation process described in section 3.8.

3.8 Required Permissions

Due to the varied experience and familiarity with the ECS system of users, each request to connect to ECS via the ECS APIs will be handled individually. After deciding that a user wishes to pursue this route, he contacts his User Services representative at the appropriate DAAC. The decision regarding approval to connect to ECS via the APIs rests with the DAAC manager.

Issues such as operational security and testing of the user's software will be negotiated at that time.

3.9 Advertising the Application

A user may wish to advertise the new capability created by his value added application. The interface that the user uses to submit advertisements to ECS is the World Wide Web. He will access an ECS Advertising Server to submit or modify advertisements. The ECS Advertising Server is maintained by ECS. The connection between the user's Web browser and the Advertising Server is made using the available internet connections.

3.9.1 ECS Advertising Service

The Advertising Information is sent from the user to the ECS. Its purpose is to provide information sufficient to allow another ECS user to locate data and services located at the application creator's location. The ECS Advertising Service will utilize HTML protocols and will accept advertisements via an interactive HTML-form based interface.

The application creator will be able to initiate an ECS Advertising Service session and link to the Advertisement Submission Form. From this form, he will be able to submit new advertisements or modify existing advertisements. He will provide information such as description of the application being advertised or data products created from it, along with information on access restriction, pricing, and copyright limitation. Product descriptions will include items such as temporal and geographic coverage, processing level, sensor, and parameter inputs. The name, address, phone numbers, and e-mail address to contact regarding the advertisement will also be entered. The creator may also submit graphical icons or logos, and Universal Reference Location (URL) links.

When he presses the submit button, the contents of the form will be assembled into a data block. This data block is posted to the Advertising Service URL where it is processed. ECS will receive the advertisement and, after reviewing and approving the advertisement, will send the application creator via e-mail either a confirmation that the advertisement has been posted to the Advertising Service or a statement that the advertisement has been rejected and the reason for the rejection.

3.10 ECS User Interface Style Guide

An ECS user developing an application with the intention of offering that service to the user community as a whole should consider using the ECS User Interface Style Guide as guidance in developing the user interface. The Guide is a working document that provides standards for designing and implementing ECS user interfaces to guide ECS developers in the creation of effective, user-friendly interfaces. Consistent application of these standards will help ensure a common "look and feel" across ECS user interfaces. By implementation of these guidelines, a more seamless integration of the application interface and the ECS interface can be accomplished.

4. Interface Services

This section describes a set of interface services which ECS users may integrate into their application programs in order to allow these programs to interact with ECS. The set of services described within these subsections is not intended to be exhaustive. These services may be expanded or modified based on further analysis of ECS requirements and user feed back. These subsections will evolve and be updated to reflect the "as-built" documentation. Each subsection addresses a specific service. It describes the process by which a service is accomplished in three ways -- narrative, code fragments, and in an event trace diagram. These three methods are used to reinforce each other.

Event Trace Diagrams

An event trace diagram, commonly referred to as an "event trace", shows a particular series of interactions among objects in a single execution of a system. An event trace diagram is drawn as follows: Objects in a transaction are drawn as vertical lines. An event is drawn as a labeled horizontal arrow from the sending object's line to the receiving object's line. The label is the method being invoked by the sending object. Time proceeds vertically, so event timing sequences can be easily determined. Additionally, an object can send simultaneous events to other objects. The following example gives a brief tutorial in reading and understanding event trace diagrams used in this document.

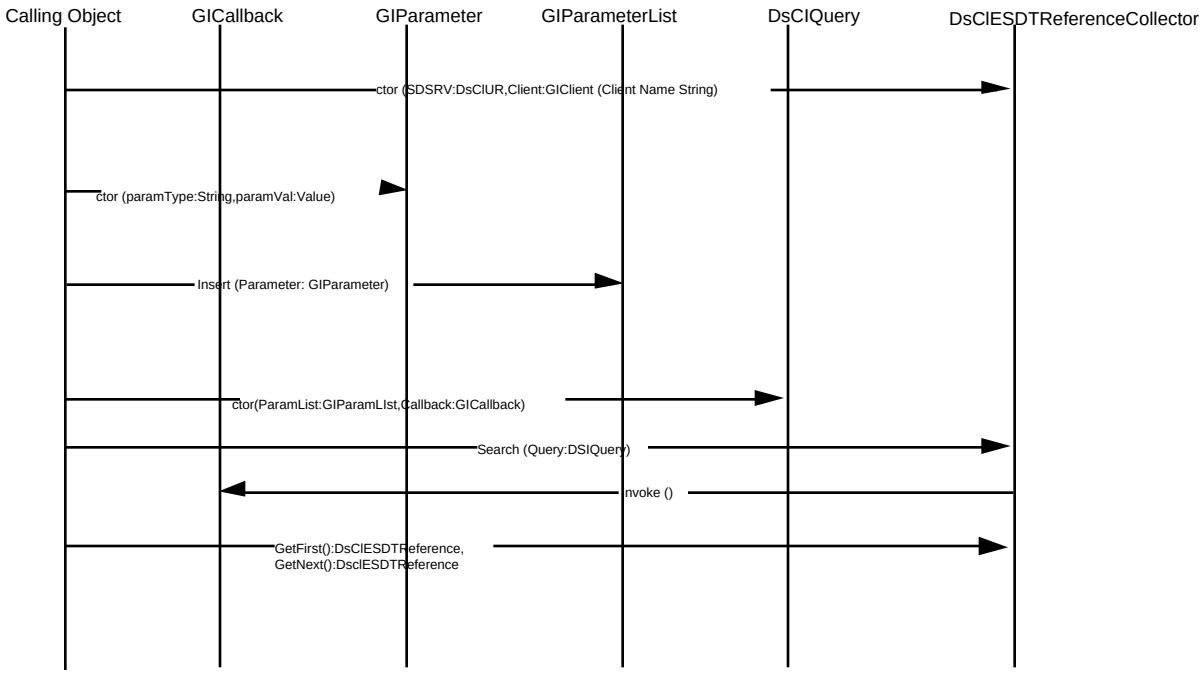

Figure 4-1 Sample Event Trace

Beginning at the top of the event trace there are seven lines and thus seven calls or events. The sequence of events flows from top to bottom on the diagram.

1) The first step of this event trace is for the calling object, to call the DsClESDTReferenceCollector class to create a Collector object with SDSRV and Client as parameters.

2) The next step is to create a GlParameter object by calling GlParameter constructor using paramType and paramVal as the parameters.

3) The parameter created in step 2 is then inserted into the GlParameterList via a call to the GlParameterList class.

4) The GlParameterList is in turn used to create a query through a call to the DsClQuery class. The GlParameterList and callback are provided as parameters.

5) Once the Query is created, the Search can be instantiated through a call to the DsClESDTReferenceCollector class.

6) After the search is completed, the DsClESDTReferenceCollector returns control to the calling object using the callback parameter that was supplied in Step 4.

7) The final step is to use the class methods within the DsClESDTReferenceCollector class, GetFirst() and GetNext(), to extract the results from the DsClESDTReferenceCollector object.

A more detailed discussion of event trace diagrams is found in "Object-Oriented Modeling and Design", Rumbaugh et al.

4.1 Query for Data

4.1.1 Query for Data using the Distributed Information Manager

4.1.1.1 Introduction

Interfacing ECS Subsystem: Distributed Information Manager (DIM) component in the Data Management Subsystem (DMS)

The purpose of this service is to provide search capabilities of EOSDIS and non-EOSDIS data, independent of the underlying ECS and non-ECS components which will be searched. The application program will provide a query in the form of the Earth Science Query Language (ESQL). ECS has selected the Illustra DBMSs SQL language as the ESQL. This section provides an example ESQL query.

The ESQL will be parsed by the DIM and subparts will be sent to the underlying components such as the Local Information Manager (LIM), the Science Data Server (SDSRV), and the V0 Gateway (GTWAY). The same search interface described in this section is also applicable to sending searches to the LIM or GTWAY. A search would be directed in this manner for performance reasons, or to restrict the result set to a single site or V0 component without explicitly specifying the site in the query.

4.1.1.2 Classes / Member Functions Used

Table 4.1-1 gives the classes and specific member functions used to **Query for Data at a DIM**.

Table 4.1-1. Class Name and Member Functions for Query for Data at a DIM (1 of 2)

Class Name and Description	
DmImClRequestServer include file : "DmImClRequestServer.h" This object is used to make a connection to the DIM server. Once connected, this objects is used to initiate asynchronous requests to the server. The DmImClRequestServer object itself works synchronously, meaning the calling object must wait for a response before continuing. This server class is inheriting from EcCsRequestServer within the Server Request Framework (SRF) Key Mechanism.	
Member Function Name	**Member Function Description**
DmImClRequestServer(server :EcUrUr, user :MSSUserProfile &)	The constructor expects the Universal Reference (UR) of the DIM, LIM, or GTWAY to which it will connect. It also expects the user information supplied in the MSSUserProfile in order to check access control lists (used for security checks) or the user as well as to keep user session information at the server side.
NewRequest(request :DmImClRequest *, requestT :RequestType)	The caller initiates the creation of a new request using this method. A DmImClRequest object will be returned in the request parameter after the associated server-side request object is created. This DmImClRequest object will be used later by the application program to submit an asynchronous request. The requestT parameter specifies the request type that will be initiated. This will specify the value QUERY, BROWSE, ACQUIRE, or other service types as specified in the Advertisement for the service (see Sections 4.5 and 4.6).

Table 4.1-1. Class Name and Member Functions for Query for Data at a DIM (2 of 2)

Class Name and Description	
DmImClRequest include file : "DmImClRequest.h" This class will handle a specific request to the DIM, LIM, or GTWAY servers. Once a request is issued and fully satisfied, this object can be reused to initiate other requests of the same type to the server. Each request is handled asynchronously, with status of the request being returned to the calling object through a callback function. This class inherits from EcCsAsynchRequest_C from the SRF Key Mechanism.	
Member Function Name	Member Function Description
SetSearchConstraints(constraints :RWCString) : EcUtStatus	This method will accept RWCString as the search constraint and pass the argument to the server. This just sets the constraint in the client side object. The request is not submitted to the server until the Submit method is called. It returns a status.
SetCallBack(DmImCallBack *) : void	Allows the application program that created the request to be notified when a state change happens within the request. A state change for example will occur when the request has completed its query and the results are available. The function supplied has to accept two parameters, myUR : EcUrUr and myState, an enumerated type inherited from SRF. myUR will be used by the caller to identify which request is calling back so that a single callback could potentially be used for multiple requests.
Submit() : EcUtStatus	When the application program invokes Submit, the request will encapsulate all commands or constraints into a message object and ship that message object to the server side where it will be processed. The message object is determined by the request type and it is shipped using the EcCsMsgHandler object (SRF).
GetResults(startpoint :int, endpoint :int) : GlParameterList	This method will allow the application program to retrieve search results . Specified are the startpoint and the endpoint which allow for a range of results to be returned. This allows callers to customize the size of the results to the application program's hardware configuration.

4.1.1.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user application must be connected to DCE (see section 3.5.1 DCE Communications)
- user application must be written in C++ to send calls to the C++ objects
- user application must use UR mechanism to point to correct DIM. The UR can be hard coded or retrieved from the Advertising Service prior to starting this scenario.
- user application must have instantiated an MSS User Profile object to identify the user using this service. If the user profile is passed as null, the system will assume guest privileges.

4.1.1.4 End Conditions

The scenario terminates with the following end conditions:

- user application called back by DIM when operation complete
- DIM DmImClRequest object returns results to client program.

4.1.1.5 Detailed Process Steps

The Query for Data at a DIM is accomplished in seven main steps, which are described below. The series of steps is also documented in the event trace in Figure 4.1-1. An explanation of how to use the event trace is given in the introduction to Section 4.

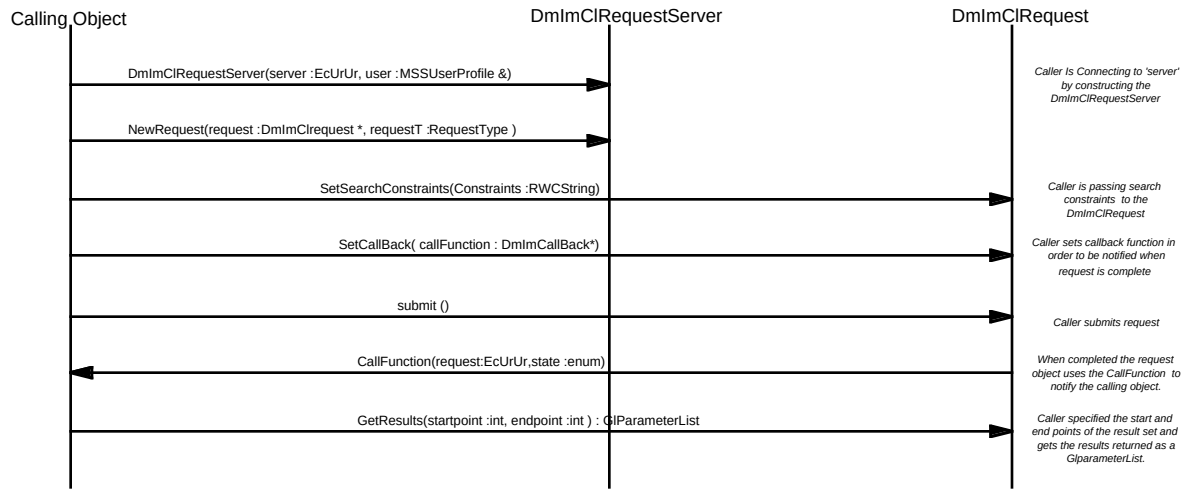

Figure 4.1-1 Query for Data Using the DIM Event Trace

Step (1) the calling object initiates a session with the DIMGR by creating a DmImClRequestServer object. The DmImClRequestServer object is a generic request factory object which establishes a connection to a server and instantiates a server-side request factory object. The server connection is established to the server with the UR specified in the constructor.

```
// retrieve user profile information from MSS
MSSUserProfile      user(TBD-11);
// retrieve server UR from advertising
EcURUR       serverUR = ad.GetUR();
DmImClRequestServer server(user, serverUR);
```

Step (2) the calling object initiates the creation of a new request by using the method NewRequest from the DmImClRequestServer object. It needs to pass a pointer to that object. That pointer will be assigned to Null until the request is build. The pointer will then be assigned to that request and is ready for use. myRequestType will be used to differentiate between Query, Browse, or Acquire.

```
DmClRequest   request;
RequestType   type = QUERY;
EcUtStatus    status;
status = server.NewRequest(&request, type);
```

Step (3) the calling object populates the DmImClRequest with the search constraints.

```
RWCString     query = “TBD-11”;
request.SetSearchConstraints(query);
```

Step (4) the calling object specifies a callback function to the request object so that the calling object can be notified of the completion of the request.

```
DmImCallback myCallback (EcURUr myUR, EcCsState state)
{
       //Process the request
}
request.SetCallBack(*(myCallback))
```

Step (5) the calling object submits the request.

```
EcUtStatus    status;
status = request.Submit();
```

Step (6) the calling object's callback is invoked upon completion of the data search.

```
//SRF transparently calls myCallback on every state change.
```

Step (7) the calling object then specifies the range of results it can accommodate and retrieves the results specified from the DmImClRequest object. For example, the client can specify that it wants results 1 through 100 returned to reduce the amount of data being returned to the user. The return type is a GlParameterList that contains the attributes as requested in the query.

```
DmImCallback myCallback (DmImClRequest *request, EcCsState state)
{
       if (state == COMPLETE)
       {
              GlParameterList     list;
              list = request->GetResults(1, 100);
              // process the results list.
       }
}
```

4.1.2 Query for Data using the Science Data Server

4.1.2.1 Introduction

Interfacing ECS Subsystem: Science Data Server (SDSRV) component in the Data Server Subsystem (DSS)

The purpose of this service is to provide an interface to search the ECS Science Data Server metadata database in order to identify the set of data that matches the provided search criteria. The user provides the search criteria in the form of a Parameter = Value list. The user queries the Science Data Server (SDSRV) for data that matches a given search criteria (e.g., spatial search, sub select, etc.). The SDSRV Query is executed and results are returned as Earth Science Data Type (ESDT) object references containing the metadata related to the relevant granules.

4.1.2.2 Classes / Member Functions Used

Table 4.1-2 gives the classes and specific member functions used to **Query for Data between DAACs**.

Table 4.1-2. Class Name and Member Functions for Query for Data between DAACs (1 of 6)

Class Name and Description	
DsClESDTReferenceCollector include file : "DsClESDTReferenceCollector.h" This public, distributed class is a specialization of the Collector class which handles DsClESDTReferences. This class provides the normal operations for ESDTReferences, the ability to handle requests, working-collection synchronization, and sessions. It also contains private operations that hand the ESDTReference-level actions to the data server	
Member Function Name	Member Function Description
DsClESDTReferenceCollector (server :DsShESDTUR &, client :GlClient &)	The constructor expects the Universal Reference (UR) of the server and the client to which it will connect.
Search (query constraints :DsClQuery &)	The Search method expects the query constraints embedded in the DsClQuery object.

Table 4.1-2. Class Name and Member Functions for Query for Data between DAACs (2 of 6)

Class Name and Description	
GlStringP include file : “GlStringP.h”, “GlParameter.h”, “GlAll.h” This public class allows the capture of the command list or the results list	
Member Function Name	Member Function Description
GlStringP (value : RWCString, name : RWCString)	The constructor creates an GlStringP object taking the name and value as specified

Table 4.1-2. Class Name and Member Functions for Query for Data between DAACs (3 of 6)

Class Name and Description	
GlParameterList include file : “GlParameterList.h” This public class allows the capture of the command list or the results list	
Member Function Name	Member Function Description
GlParameterList ()	The constructor creates an empty GlParameterList object
at (i)	The at() method allows access to the individual GlParameters in a GlParameterList at the ith entry on the list.
insert (parameter : GlParameter &)	The insert() method allows insertion of the individual GlParameters into a GlParameterList at the next entry in the list.

Table 4.1-2. Class Name and Member Functions for Query for Data between DAACs (4 of 6)

Class Name and Description	
DsClQuery include file : “DsClQuery.h” This public, local class simplifies the passing of query information from the client to the data server. The object is created in client space. The contents of the object will be used to create a request object which will be passed to the data server. It is assumed that the “from” clause of an SQL query is inherent in specification of the data server to which the query is issued, i.e. that the query is against the inventory of the data server. Any conversion to actual table names which may be necessary is done transparently to the client software.	
Member Function Name	Member Function Description
DsClQuery (constraints :GlParameterList *)	The constructor expects the constraints which are normally specified in the “from” clause.

Table 4.1-2. Class Name and Member Functions for Query for Data between DAACs (5 of 6)

Class Name and Description	
GlCallback include file : "GlCallback.h" This public class allows the status of the request to be made available to the client	
Member Function Name	Member Function Description
GlCallback ()	The constructor creates an empty GlCallback object

Table 4.1-2. Class Name and Member Functions for Query for Data between DAACs (6 of 6)

Class Name and Description	
ECUtStatus include file : "EcUtStatus.h" This public class is a utility class to capture the return status following a method call or similar action.	
Member Function Name	Member Function Description
EcUtStatus ()	The constructor constructs an empty class.
Ok()	Used to verify the return status is EcUtStatus::OK

4.1.2.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user application must be connected to OODCE (see section 3.5.1 DCE Communications)
- user application must be written in C++ to send calls to the C++ objects
- user application must use UR mechanism to point to correct SDSRV

4.1.2.4 End Conditions

The scenario terminates with the following end conditions:

- user application called back by SDSRV when operation complete
- SDSRV ESDT Reference Collector objects (DsClESDTReferenceCollector) contain query results

4.1.2.5 Detailed Process Steps

The Query for Data between DAACs is accomplished in nine main steps. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

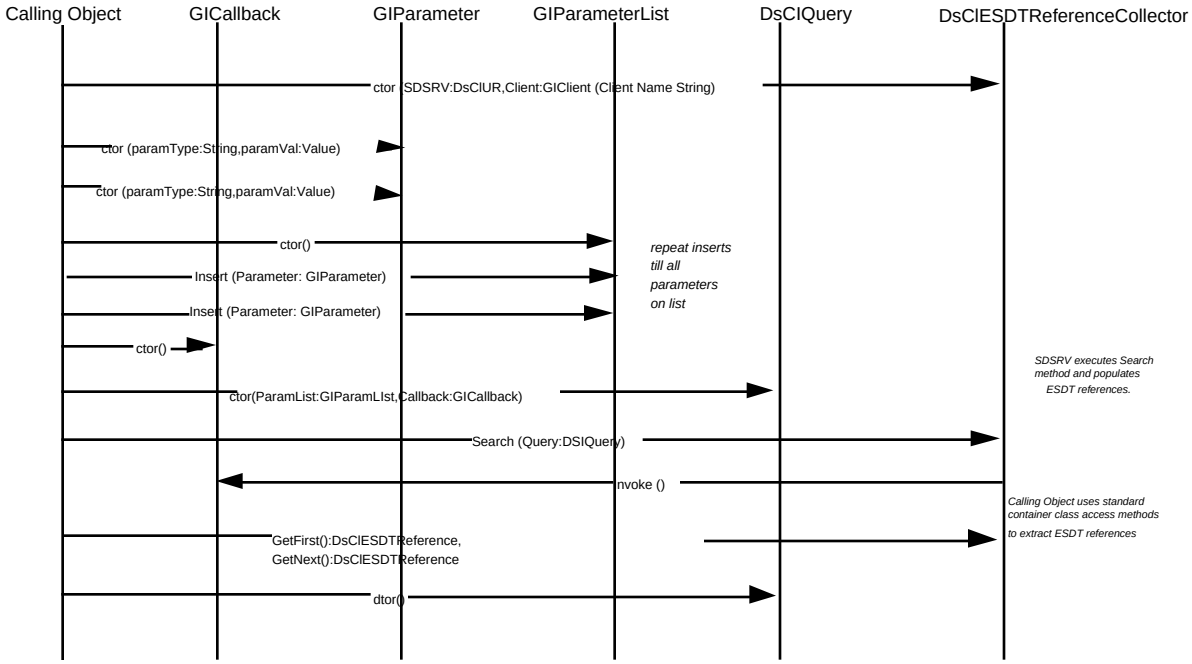

Figure 4.1-2 Query for Data Using the Science Data Server Event Trace

Step (1) an instance of the class DsClESDTReferenceCollector is constructed by the Calling Object. This step is represented by the first (top) event in the Event Trace.

```
EcUtStatus collectorStatus;

DsClESDTReferenceCollector* collector =

DsClESDTReferenceCollector::Create( collectorStatus,

                                    DsShSciServerUR( Init( "DataServerName")),

                                    GlClient("ClientName"));
```

Step (2) an instance of the class GlParameter is constructed. The object accepts a parameter type (String) and a parameter value (Value). In the case of the search in this example two parameters types are passed. This step is represented by the next two events in the Event Trace.

```
GlStringP startDateConstraint = RWCString("1/1/93") ;

dateConstraint.SetName( "startDate") ;

dateConstraint.SetDescription( ">") ;
```

```
GlStringP endDateConstraint = RWCString("1/15/93") ;

dateConstraint.SetName( "endDate") ;

dateConstraint.SetDescription( "<") ;
```

Step (3) an instance of the class GlParameterList is created by the Calling Object and a parameter (GlParameter) is inserted into it. In this example, two parameters are inserted into it. This step is represented by the next three events in the Event Trace.

```
GlParameterList constraints( "Constraints");

constraints.insert( &startDateConstraint) ;

constraints.insert( &endDateConstraint) ;
```

Step (4) an instance of GlCallback is created by the Calling Object. This step is represented by the next event in the Event Trace.

```
GlCallback myCallback;
```

Step (5) the calling operation constructs a query by creating an instance of DsClQuery. The query object accepts a parameter list (GlParameterList) and a callback function (GlCallback) which is invoked when the query is complete. The parameter list is a set of parameter/value pairs which are combined to define the search criteria. The search criteria is the conjunction of spatial, temporal, and keyword parameters. This step is represented by the next event in the Event Trace.

```
EcUtStatus returnStatus ;

DsClQuery* firstQuery = DsClQuery::Create( returnStatus, &constraints) ;
```

Step (6) the search is performed by the Calling Object passing the query (DsClQuery) to the instance of DsClESDTReferenceCollector. This step is represented by the next event in the Event Trace.

```
EcUtStatus searchStatus ;



searchStatus = collector->Search(DsClQuery&);
```

Step (7) the instance of DsClESDTReferenceCollector invokes the instance of GlCallback, which notifies the Calling Object that the search is complete. This step is represented by the next event in the Event Trace. No action is required on the part of the user; this operation occurs within ECS.

Step (8) the Calling Object uses standard container class access methods (GetFirst, GetNext) to extract the ESDT references. This step is represented by the next event in the Event Trace.

```
GetFirst();

GetNext();
```

Step (9) the Calling Object destructs the instance of the DsClQuery class. This step is represented by the final (bottom) event in the Event Trace.

```
DsClQuery::~DsClQuery()
```

4.2 Ordering and Receiving Data

4.2.1 Ordering and Receiving of Data using the Distributed Information Manager

4.2.1.1 Introduction

Interfacing ECS Subsystem: Distributed Information Manager (DIM) component in the Data Management Subsystem (DMS)

The purpose of this service is to provide ordering capabilities on EOSDIS and non-EOSDIS data, independent of the underlying ECS and non-ECS components which will be searched. It is assumed that a search has been performed and the application program can identify specific URs for granules based on the results of the search. The application program will submit the commands to "acquire" the data. If the application program waits for completion status from the DIM, the data can be received as well. If the application program disconnects, then an e-mail notification will be sent to the user specifying that the data is ready for pick up or has been distributed.

4.2.1.2 Classes / Member Functions Used

Table 4.2-1 gives the classes and specific member functions used to **Order and Receive Data at a DIM**.

Table 4.2.1-1. Class Name and Member Functions Order and Receive Data at a DIM (1 of 5)

Class Name and Description	
DmImClRequestServer include file : "DmImClRequestServer.h" This object is used for creating requests and to send session command to the server side. It is synchronously connected to the server and its UR is used as a session Id.	
Member Function Name	Member Function Description
DmImClRequestServer(server :EcUrUR, user :MSSUserProfile &)	The constructor expects the Universal Reference (UR) of the DIM, LIM, or GTWAY to which it will connect. It also expects the user information supplied in the MSSUserProfile in order to check access control lists (used for security checks) or the user as well as to keep user session information at the server side.
NewRequest(request :DmImClrequest *, requestT :RequestType)	The caller initiates the creation of a new request using this method. A DmImClRequest object will be returned in the request parameter after the associated server-side request object is created. This DmImClRequest object will be used later by the application program to submit an asynchronous request. The requestT parameter specifies the request type that will be initiated. This will specify the value QUERY, BROWSE, ACQUIRE, or other service types as specified in the Advertisement for the service (see Sections 4.5 and 4.6).

Table 4.2.1-1. Class Name and Member Functions Order and Receive Data at a DIM (2 of 5)

Class Name and Description	
DsClCommand include file : “DsClCommand.h” A specialization of the DsClCommand class for client interfaces. Adds constructors that ease building of commands based on advertisements, or special direct commands that are "built- in" to the data server and do not correspond to advertisements.	
Member Function Name	Member Function Description
DsClCommand	Used to construct a command from its basic parts: service name, parameters, and category.

Table 4.2.1-1. Class Name and Member Functions Order and Receive Data at a DIM (3 of 5)

Class Name and Description	
GlParameter include file : “GlParameter.h” This is an abstract base class that represents a single parameter that can be passed to many ECS objects. A parameter has a name and an optional description, as well as a value which depends upon its type. Parameters are usually collected together (GlParameterList), and used to dynamically specify values for service calls, result lists, etc. There are several types derived from GlParameter, which implement the value() member function to return an appropriately typed value.	
Member Function Name	Member Function Description
GlParameter (char *) : void	Constructs a parameter with the given name.

Table 4.2.1-1. Class Name and Member Functions Order and Receive Data at a DIM (4 of 5)

Class Name and Description	
GlParameterList include file : “GlParameterList.h” This class represents a collection of parameters, and is itself derived from GlParameter. Therefore, GlParameterLists can be embedded in themselves to any depth.	
Member Function Name	Member Function Description
GlParameterList	Constructs an empty, unnamed parameter list.
insert(GlParameter)	Inserts parameters into GlParameterList object.

Table 4.2.1-1 Class Name and Member Functions Order and Receive Data at a DIM (5 of 5)

Class Name and Description	
DmImClRequest include file : "DmImClRequest.h" This class will handle all requests submitted by the caller . It is part of the Server Request Framework by inheriting from EcCSAynchRequest_C.	
Member Function Name	Member Function Description
AddCommand(oneCmd :DsClCommand) : EcUtStatus	Will allow the caller to add a command to the list of commands contained in myCommandList. The entire set is passed to the message class when submit is called.
SetCallBack(DmImCallBack *) : void	Allows the application program that created the request to be notified when a state change happens within the request. A state change for example will occur when the request has completed its query and the results are available. The function supplied has to accept two parameters, myUR : EcUrUr and myState, an enumerated type inherited from SRF. myUR will be used by the caller to identify which request is calling back so that a single callback could potentially be used for multiple requests.
Submit() : EcUtStatus	When the application program invokes Submit, the request will encapsulate all commands or constraints into a message object and ship that message object to the server side where it will be processed. The message object is determined by the request type and it is shipped using the EcCsMsgHandler object (SRF).
GetResults(startpoint :int, endpoint :int) : GlParameterList	This method will allow the application program to retrieve search results . Specified are the startpoint and the endpoint which allow for a range of results to be returned. This allows callers to customize the size of the results to the application program's hardware configuration.

4.2.1.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user application must be connected to a DIM and already retrieved search results. The application uses the same DmClRequestServer to initiate the Acquire request.
- user application must be written in C++ to send calls to the C++ objects

4.2.1.4 End Conditions

The scenario terminates with the following end conditions:

- user application called back by DIM when operation complete, if application program keeps connection open.

4.2.1.5 Detailed Process Steps

The Ordering and Receiving of Data at a DIM is accomplished in eleven main steps, which are described below. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

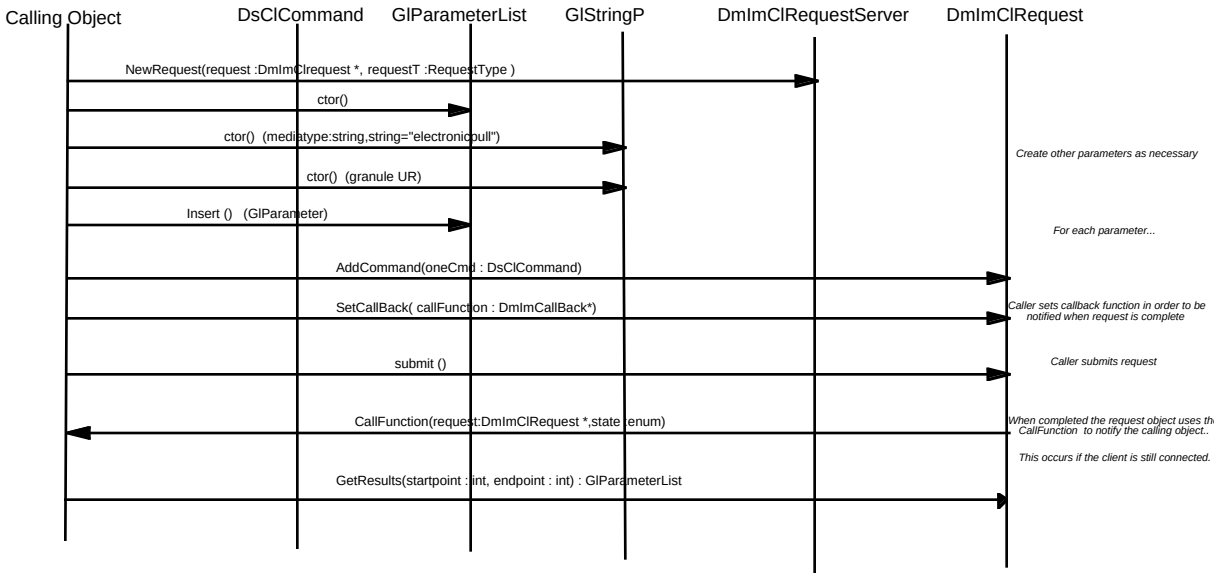

Figure 4.2.1-1. Ordering and Receiving of Data Event Trace

Step (1) the calling object initiates the creation of a new request by using the method NewRequest from the DmImClRequestServer object. It needs to pass a pointer to a DmImClRequest object. Once a DmImClRequest object is created it will be copied to the pointer passed into the NewRequest method. myRequestType will be used to differentiate between Queries, Browse, or Acquire.

```
// the server variable was established during the search phase and
// is of type DmImClRequestServer.
DmClRequest  request;
RequestType  type = ACQUIRE;
EcUtStatus   status;
status = server.NewRequest(&request, type);
```

Step (2), the next four lines in the event trace, the calling object constructs a GlParameterList which contains the parameters that need to be specified for the Acquire request. These will be used to construct a DsClCommand object that will be submitted with the request.

```
// Create an empty list that will contain the parameters of
// the acquire command.
GlParameter list("Acquire Parameters");
```

```
// The actual possible values of the media type and format
// parameters would be contained in the Advertising Service.
// The application program could supply them as string variables
// to the GlStringP constructor.
GlStringP    mediaType("ftpPull", "media_type");
list.insert(&mediaType);
GlStringP    mediaFormat("Compressed, HDF", "media_format");
list.insert(&mediaFormat);

// It is assumed that multiple granule URs would be contained in
// a list at the end of the options.
GlParameterList     urlist("UR List");
for (i = 0; i < num_granules; i++)
{
       GlStringP    ur(urstrings[i]);
       urlist.insert(&ur);
}
// A parameter of a list can be a list itself to support
// complex parameters.
list.insert(&urlist);
```

Step (3), the next line of the event trace, the calling object initiates the creation of a DsClCommand. A list of parameters which define the data to be acquired is inserted into the DsClCommand object, and an attribute of DsClCommand is set to indicate that it is an Acquire command.

```
// Construct the command object given the parameters created
// in Step 2.
DsClCommand  cmd(DsCGeAcquire, list);
```

Step (4) the calling object populates the DmImClRequest with the DsClCommand.

```
// Add the command object to the DmImClRequest.  In real life,
// there could be multiple commands, such as subset and then
// acquire.  If this were needed, then the AddCommand method,
// would be called for each DsClCommand object created.
request.AddCommand(cmd);
```

Step (5) the calling object specifies a callback function to the request object so that the calling object can be notified of the completion of the request.

```
DmImCallback myCallback (EcURUr myUR, EcCsState state)
{
       //Process the request.
}

request.SetCallBack(*(myCallback));
```

Step (6) the calling object submits the request.

```
EcUtStatus   status;
status = request.Submit();
```

Step (7) the calling object's callback is invoked upon completion of the data acquire. The client can disconnect from the server if necessary, before the acquire command is totally complete. For example, if a media order is specified, then the acquire may take hours to complete. If the client disconnects, then the callback will not be initiated. If the client is not present at the completion of the acquire, the user will receive direct notification through e-mail that the acquire has been completed. If the client does stay connected, then the callback will be initiated and the connection can be terminated.

```
//SRF transparently calls myCallback on every state change.
```

Step Eight (8) the calling object can initiate an FTP session given that it knows that the request has been completed. The results of the acquire should be the FTP location to pull from.

```
   // Create a Global Resource pointer

   DsStResourceProvider* resource = 0;


   // Create a return status and class GlParameterList to contain

   FTP username, password, host, pull source

   EcUtStatus stat;
   GlParameterList* parms = new GlParameterList();


   // Create an instance of the DsStStagingDisk class

   RWCString myStagingDiskName;

   const double disksize = 700;
   RWCString hostname("disk Server hostname");

   DsStStagingDisk* mydisk = DsStStagingDisk::Create(hostname,
                                              disksize,
                                             (long)0);
   myStagingDiskName = mydisk->GetStagingDiskName();

   // Insert FTP username, password, host, pull source names into
GlParameterList prior to execution of
   ftp pull command
const RWCString& tag = DsCDdFTPPULL;
   parms->insert(new GlStringP ("smalladi", DsCDdFTPUSER));
   parms->insert(new GlStringP ("sri1234", DsCDdFTPPASSWORD));
   parms->insert(new GlStringP ("kodiak", DsCDdFTPHOST));
   parms->insert(new GlStringP ("Disk::2", DsCDdFTPPULLSOURCE));

   // Perform ftp pull comand

   stat = resource->Exec(tag, parms);

   // Check return status

   if(!stat.Ok())
   {
         delete parms;
          mydisk->Destroy();
         resource->Destroy();
   }
```

4.2.2 Ordering and Receiving of Data using the Science Data Server

4.2.2.1 Introduction

Interfacing ECS Subsystem: SDSRV

The purpose of this service is to provide the user the ability to acquire data directly from the Science Data Server.

4.2.2.2 Classes / Member Functions Used

Table 4.2-2 gives the classes and specific member functions used to Order and Receive Data using the Science Data Server.

Table 4.2.2-1. Class Name and Member Functions Order and Receive Data at the DataServer (1 of 5)

Class Name and Description	
DsClESDTReferenceCollector include file : "DsClESDTReferenceCollector.h" This public, distributed class is a specialization of the Collector class which handles DsClESDTReferences. This class provides the normal operations for ESDTReferences, the ability to handle requests, working-collection synchronization, and sessions. It also contains private operations t hand the ESDTReference-level actions to the data server	
Member Function Name	Member Function Description
DsClESDTReferenceCollector (server :DsShESDTUR &, client :GlClient &)	The constructor expects the Universal Reference (UR) of the server and the client to which it will connect.

Table 4.2.2-1. Class Name and Member Functions Order and Receive Data at the DataServer (2 of 5)

Class Name and Description	
DsClCommand include file : "DsClCommand.h" This public, class is a specialization of the DsCommand for client interfaces. Adds constructors that ease the building of commands based on advertisements, or special direct commands that are "built-in" to the data server and do not correspond to advertisements. The commands are constructed by use of the GlParameterList Class	
Member Function Name	Member Function Description
DsClCommand (service :RWCString &, ParamList :GlParameterList, commandCategory :DsEShSciCommandCatagory &)	The constructor expects the service name the command parameters and the command category.
SetFor8mmTape(mediaFormat :RWCString &, fileFormat :RWCString &)	The caller set the desired action prior to issuing the command. The media and file formats are needed to specify the parameters prior to an acquire to 8mm tape.

Table 4.2.2-1. Class Name and Member Functions Order and Receive Data at the DataServer (3 of 5)

Class Name and Description	
GlCallback include file : "GlCallback.h" This public class allows the status of the request to be made available to the client	
Member Function Name	Member Function Description
GlCallback ()	The constructor creates an empty GlCallback object

Table 4.2.2-1. Class Name and Member Functions Order and Receive Data at the DataServer (4 of 5)

Class Name and Description	
DsClRequest include file : "DsClRequest.h" This public class is a specialization of the DsRequest for client interfaces. Allows the client to compose a request and submit it to the data server. Once submitted, the status may be polled, or a callback can be provided that is triggered on every status change.	
Member Function Name	Member Function Description
DsClRequest (command :DsClCommand &, priority :DsEShSciPriority &)	The constructor expects the commands constructed by use of the GlParameterList Class and the optional command priority (Default is NORMAL).
Submit(collector/dataServer :DsClESDTCollector *, domain :DsTShRequestDomain &)	Used to submit a request to be executed by a single, specific ESDT. The request is in turn submitted to the "implied" DsClESDTReferenceCollector. i.e. the one the DsClESDTReferenceCollector holds a pointer to. Optionally the request may be submitted with a domain. (Default is rwnil)

Table 4.2.2-1. Class Name and Member Functions Order and Receive Data at the DataServer (5 of 5)

Class Name and Description	
GlParameterList include file : "GlParameterList.h" This public class allows the capture of the command list or the results list	
Member Function Name	Member Function Description
GlParameterList ()	The constructor creates an empty GlParameterList object
at (i)	The at() method allows access to the individual GlParameters in a GlParameterList at the ith entry on the list.

4.2.2.3 Start Conditions/Initialization

The scenario assumes the following starting conditions:

- client application must be connected to a Science Data Server and already received

The application uses the same DsClESDTReferenceCollector to maintain the connection.

- user application called back by SDSRV when operation complete
- user application must be written in C++ to send calls to other C++ objects

4.2.2.4 End Conditions

The scenario terminates with the following end conditions:

- user application called back by Science Data Server when operation complete

4.2.2.5 Detailed Process Steps

The Ordering and Receiving of Data using the Science Data Server is accomplished in the following 7 steps, which are described below. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

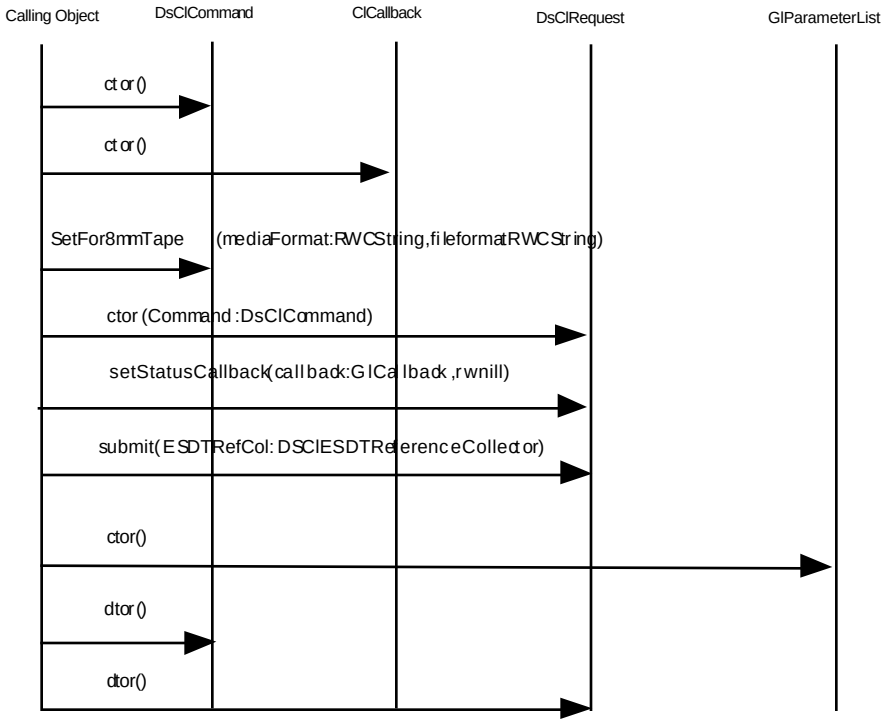

Figure 4.2.2-1. Ordering and Receiving of Data using the SDSRV event trace

This assumes that an instance of the class DsClESDTReferenceCollector already exists from the previous query session. It may have been instantiated in the following manner, where HostName is the server UR:

```
GlClient    theClient("AnyUser");
GlUR        theServer("HostName");
DsClESDTReferenceCollector ESDTRefCol(theServer, theClient);
```

Assume it has been updated with the UR of the granule to acquire during the previous query.

Step(1) an instance of the class DsClCommand is created by the calling object and the RWCStrings representing the service parameters are inserted into it. In this example one service parameter is inserted into it. This action is represented by the next two steps in the Event Trace.

```
// Create service parameters in RWCStrings

RWCString mediaformat("ExampleMediaFormat");
RWCString fileformat("ExampleFileFormat");

// Create acquire command object
DsClCommand* myCommand = new DsClAcquireCommand();
```

Step(2) the target for the acquire may be set by the Calling Object. There are options depending on the desired action. Choices are Set for FTP pull, set for FTP push or set to 8mm tape, as performed in the example.

```
myCommand->SetFor8mmTape(mediaformat,fileformat);
```

Step(3) an instance of the GlCallback object is constructed.

```
GlCallback myCallback;
```

Step(4) an instance of the DsClRequest object is instantiated

```
DsClRequest myRequest(*myCommand);
```

Step(5) the request is submitted to the server by using the Submit method with the domain specified

```
myRequest.Submit(ESDTRefCol);
```

Step(6) the instance of the DsClReferenceCollector invokes the instance of GlCallback, which notifies the Calling Objects that the search is complete. This step is represented by the next event in the Event Trace.

TBD-18

Step(7) Need to assess the results. The Science Data Server returns a parent result pointer from the request. Create three instances of GlParameterLists more needed for multiple commands. Only one command was issued in this example.

```
const GlParameterList &requestResults = myRequest.GetResults() ;

// The return result parameter list looks like this:

// Request
//    |
//    |-commandResults
//    |         |
//    |         |-scienceGroup
//    |                |
//    |                |-UR
//    |
//    |
//    |-commandResults // If we used two commands
//                      // the next set of results
                       // would be here

commandResults = (GlParameterList*)requestResults.at(0);
```

Any DsClRequest that is submitted without using SetStatusCallback is automatically synchronous. It is submitted with code similar to this:

```
EcUtStatus stat = request.Submit(collector);
```

When the Submit call returns, execution of the entire request is complete, and final status and results are immediately available. The EcUtStatus code that is returned simply indicates whether the request was successfully submitted, i.e. whether it reached the server, was validated, and passed security. (Note that this means that even if execution of one (or more) of the commands fail, the submit status of the request will probably be success.)

The full status of the request is obtained like this:

```
const GlParameterList& reqstatus = request.GetStatus();
```

This function returns a GlParameterList, named “ReqUpdate”/DsCShReqUpdatePL, that contains several parameters. Here’s a table showing what parameters may be in it:

Table 4.2.2-2. GlParameterList Parameters

Name	Type	When?	Contents
"CmdCount" (DsCShAsyncStatusCmdCountP)	GlLongP	Always	number of commands attempted
"Done" (DsCShAsyncStatusDoneP) (a only)	*GlStringP*	*Request is finished executing*	*<nothing>*
"ReqSuccess" (DsCShReqStatusP) *(b)*	GlLongP	Request is finished executing	0 = failed, non-zero = succeeded
"ReqFailReason" (DsCShReqFailReasonP) *(b)*	GlStringP	ReqSuccess == 0	reason for failure

The server executes each command in the request sequentially, and stops if any command fails. The results and status of each command is contained in the request results, which can be obtained like this:

```
const GlParameterList& reqresults = request.GetResults();
```

This parameter list (named "ReqResults"/DsCShReqResultsPL) contains a set of parameter lists, one for each command that was attempted. Each of these lists is named "CmdResults"/DsCShCmdResultsPL, and contains at least one parameter, a boolean GlLongP parameter named "CmdSuccess"/DsCShCmdStatusP. This parameter indicates whether execution of that command was successful (0 means it failed, non-zero means it succeeded). So code like this could check each of the commands:

```
for (size_t cmd = 0; cmd < reqresults.entries(); cmd++)
{
        GlParameterList* cmdresults = (GlParameterList*) reqresults[cmd];
    GlLongP* cmdstatus = (GlLongP*) cmdresults->FindParameter(DsCShCmdResultsPL);
        if (cmdstatus->value())
        {
                cout << "Command " << cmd << " succeeded!" << endl;
        }
        else
        {
                GlStringP* reason = (GlStringP*) cmdresults-
>FindParameter(DsCShCmdFailReason); // (b)
                cout << "Command " << cmd << " failed! " << reason << endl;
        }
}
```

Command result parameter lists may also contain parameters giving whatever results (specific to each particular service) were generated by executing the command. If a command fails, a GlStringP parameter in the list (named "CmdFailReason" / DsCShCmdFailReasonP *(b)*) contains text describing the reason for the failure.

Commands that execute over the collection of ESDT granules contain one more layer of parameter lists: each command results parameter list contains a set of parameter lists and status parameters for each ESDT granule upon which the command was executed. The ESDT results

parameter lists are named “ESDTResults” / DsCSrESDTResultsPL, and contain zero or more parameters that the granule returned as results from the execution of the command. After each ESDT results list will be a boolean GlLongP parameter named “ESDTStatus” / DsCSrESDTStatusP that indicates whether execution of the command upon that ESDT granule was successful (0 means it failed, non-zero means it succeeded). If the execution failed for a granule, a GlStringP parameter (“ErrorMsg” / DsCGeErrorMsg *(b)*) will hold text for the reason. (Note that even if a command fails for a particular ESDT granule, the server will continue to try to execute the command for the other ESDT granules in the collection, and the status of the command will be success).

As an example, here’s the structure of the results GlParameterList from a request that:

1) has one command, “Subset”
2) is executed on a collection of two granules
3) that succeeds for the first granule
4) that fails for the second granule

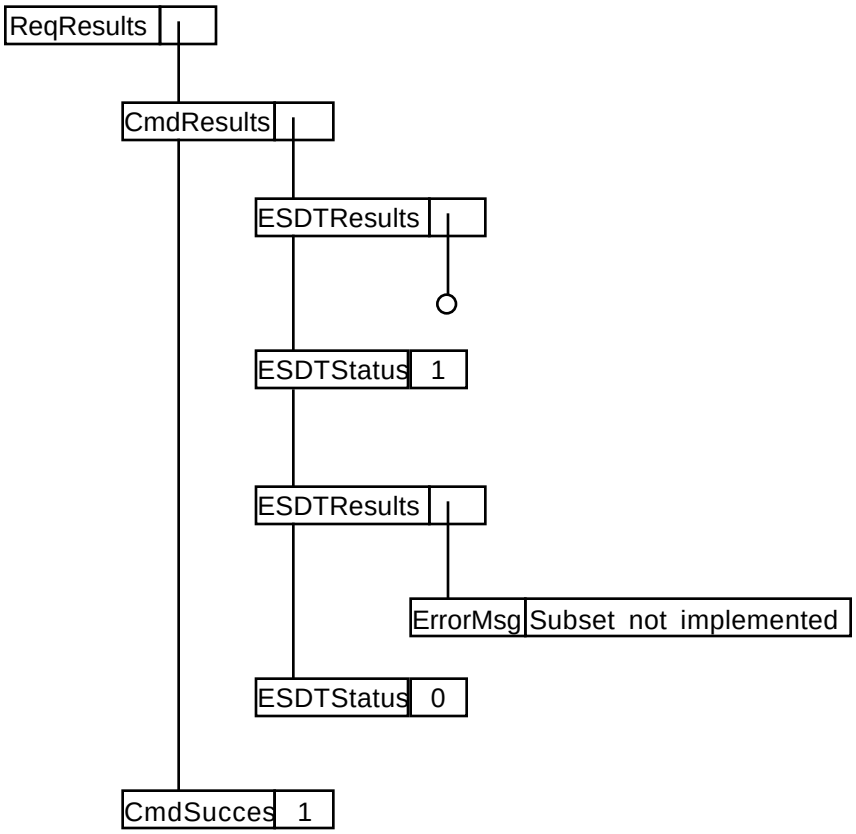

Figure 4.2.2-2. Structure of Sample Result

4.3 Order and Request Tracking

4.3.1 Introduction

Interfacing ECS Subsystem: System Management Subsystem (MSS).

The purpose of this service is to provide a means of reporting the status of orders and requests in near real time. To accomplish this, order and request information is maintained in a relational data base at the local MSS server. Applications provide status information to MSS by storing request and order information in instances of classes EcAcOrder and EcAcRequest. These objects are stored, retrieved, updated and deleted in the data base by MSS's distributed order manager represented on the client side by class EcAcOrderCMgr.

ECS requests not involving a user order such as Ingest are represented by request objects alone. User orders are represented by an order object and one or more request objects. Applications will break up an order as necessary for processing within and across DAACs. Requests can also be broken down into sub requests as necessary for processing by the applications. A parent-child relationship is maintained in requests objects in order to track all components making up an order.

Applications are responsible for creating and populating order and request objects. When a new object has been created, it is passed to the Order Manager (an instance of EcAcOrderCMgr). The Order Manager stores the object in it's data base and returns a unique ID back to the application. The Manager provides methods for updating, retrieving and deleting objects based on its ID. In the case of an initial order, the Order Manager also will perform a credit check to determine if a user has the necessary funds in his account. Additionally, the Order Manager has methods to retrieve order and request lists from the data base for the purpose of retrieving status information.

4.3.2 Classes / Member Functions Used

Table 4.3-1 list the classes and member functions used to perform **Order and Request Tracking**.

Table 4.3-1. Class Name and Member Functions for Order and Request Tracking (1 of 3)

Class Name and Description	
EcAcOrder This class defines an 'order' object that contains order specific parameters, an order ID and an order status. An application uses an 'order' object to store order parameter and to update an order's status. The application passes the 'order' object to MSS's Order and Request Tracking Server via the 'CreateOrder' method of the of the Order Manager client: EcAcOrderCMgr. At Order creation , an order ID is generated by the Order and Request Tracking Server and stored in the order object. The order ID is also passed back to the application.	
Member Function Name	**Member Function Description**
EcAcOrder()	Constructor: Creates an empty order object
GetOrder(RWCString& orderId, RWCString& userId, MsAcUsrName& userName,	Retrieves Order's Attributes

RWCString& eMailAddr, RWCString& orderStatus, RWCString& orderDesc, RWCString& orderDistFormat, RWCString& orderMedia, EcTInt& orderSize, EcTInt& orderGranule, RWCString& orderPriority, RWCString& orderHomeDAAC, MsAcAddress& shipAddr, RWCString& receiveDateTime, RWCString& startDateTime, RWCString& finishDateTime, RWCString& timeOfLastUpdate, RWCString& shipDateTime, RWCString& cancelledFlag, RWCString& abortedFlag, MsAcAddress& billingAddr, EcTFloat& price, RWCString& desiredDateTime, RWCString& estimatedDateTime, RWCString& updatedById, EcTInt& processingDAACFlags)	
RWCString& GetUserId() RWCString& GetOrderId() MsAcUsrName& GetUserName() RWCString& GetEMailAddr() RWCString& GetOrderStatus() RWCString& GetOrderDesc() RWCString& GetOrderDistFormat() RWCString& GetOrderMedia() EcTInt& GetOrderSize() EcTInt& GetOrderGranule() RWCString& GetOrderPriority() RWCString& GetOrderHomeDAAC() MsAcAddress& GetShipAddr() RWCString& GetReceiveDateTime() RWCString& GetStartDateTime() RWCString& GetFinishDateTime() RWCString& GetTimeOfLastUpdate() RWCString& GetShipDateTime() RWCString& GetCancelledFlag() RWCString& GetAbortedFlag() MsAcAddress& GetBillingAddr() EcTFloat& GetPrice() RWCString& GetDesiredDateTime() RWCString& GetEstimatedDateTime() EcTInt& GetProcessingDAACFlags() RWCString& GetUpdatedById()	Methods to retrieve individual order attributes
SetOrder(const RWCString& orderId, const RWCString& userId, const MsAcUsrName& userName, const RWCString& eMailAddr, const RWCString& orderStatus, const RWCString& orderDesc, const RWCString& orderDistFormat, const RWCString& orderMedia, const EcTInt& orderSize, const EcTInt& orderGranule, const RWCString& orderPriority, const RWCString& orderHomeDAAC,	Sets Order attributes

const MsAcAddress& shipAddr, const RWCString& receiveDateTime, const RWCString& startDateTime, const RWCString& finishDateTime, const RWCString& timeOfLastUpdate, const RWCString& shipDateTime, const RWCString& cancelledFlag, const RWCString& abortedFlag, const MsAcAddress& billingAddr, const EcTFloat& price, const RWCString& desiredDateTime, const RWCString& estimatedDateTime, const RWCString& updatedById)	
SetUserId(RWCString& userIdIn) SetOrderId(RWCString& orderIdIn) SetUserName(MsAcUsrName& userNameIn) SetEMailAddr(RWCString& eMailAddrIn) SetOrderStatus(RWCString& orderStatusIn) SetOrderDesc(RWCString& orderDescIn) SetOrderDistFormat(RWCString& orderDistFormatIn) SetOrderMedia(RWCString& orderMediaIn) SetOrderSize(EcTInt& orderSizeIn) SetOrderGranule(EcTInt& orderGranuleIn) SetOrderPriority(RWCString& orderPriorityIn) SetOrderHomeDAAC(RWCString& orderHomeDAACIn) SetShipAddr(MsAcAddress& shipAddrIn) SetShipDateTime(const RWCString& shipDateTimeIn) SetReceiveDateTime(const RWCString& receiveDateTimeIn) SetStartDateTime(const RWCString& startDateTimeIn) SetFinishDateTime(const RWCString& finishDateTimeIn) SetTimeOfLastUpdate(const RWCString& timeOfLastUpdateIn) SetCancelledFlag(const RWCString& cancelledFlagIn) SetAbortedFlag(const RWCString& abortedFlagIn) SetBillingAddr(const MsAcAddress& billingAddrIn) SetPrice(const EcTFloat& priceIn) SetDesiredDateTime(const RWCString& desiredDateTimeIn) SetEstimatedDateTime(const RWCString estimatedDateTimeIn) SetUpdatedById(const RWCString& updatedByIdIn)	Methods to set individual order attributes
SetProcessingDAAC(const RWCString& daacName)	Sets a bit indicating that the DAAC specified by daacName is processing this order. Should be called for each DAAC that is processing this order. Valid DAAC names are: Bit 0 = GSFC 1 = LARC 2 = ORNL 3 = NSIDC 4 = JPL 5 = ASF 6 = EDC

Table 4.3-1. Class Name and Member Functions for Order and Request Tracking (2 of 3)

Class Name and Description	
EcAcRequest This class defines an 'request' object that contains processing information and status for ECS requests. If an order is being processed, the order is broken into one or more requests. Requests can be broken into sub requests. A 'request' object contains request specific parameters, a request ID, a parent request ID and the request status. An application uses an 'request' object to store request parameters and to update it's status. The application passes the 'request' object to MSS's Order and Request Tracking Server via the 'CreateRequest' method of the of the Order Manager client: EcAcOrderCMgr. At Request creation , a request ID is generated by the Order and Request Tracking Server and stored in the request object. The request ID is also passed back to the application.	
Member Function Name	**Member Function Description**
EcAcRequest()	Constructor: Creates an empty request object
EcTVoid GetRequest(RWCString& orderId, RWCString& requestId, RWCString& parentId, MsAcUsrName& userName, RWCString& eMailAddr, RWCString& requestDesc, RWCString& requestStatus, RWCString& requestDistFormat, EcTInt& numFiles, EcTInt& numBytes, EcTInt& numGranule, RWCString& deviceId, RWCString& deviceDensity, RWCString& tapeFormat, RWCString& mediaType, RWCString& ESDT_Id, RWCString& requestPriority, RWCString& requestHomeDAAC, MsAcAddress& shipAddr, RWCString& receiveDateTime, RWCString& startDateTime, RWCString& finishDateTime, RWCString& timeOfLastUpdate, RWCString& shipDateTime, RWCString& FtpAddress, RWCString& FtpPassword, RWCString& destinationNode, RWCString& destinationDirectory, RWCString& requestType, RWCString& productId, RWCString& requestOwnerId, RWCString& updatedById)	Retrieves Request's Attributes
RWCString& GetOrderId() RWCString& GetRequestId() RWCString& GetParentId() MsAcUsrName& GetUserName() RWCString& GetEMailAddr() RWCString& GetRequestDesc() RWCString& GetRequestStatus() RWCString& GetRequestDistFormat() EcTInt& GetNumFiles() EcTInt& GetNumBytes() EcTInt& GetNumGranule() RWCString& GetDeviceId() RWCString& GetDeviceDensity() RWCString& GetTapeFormat() RWCString& GetMediaType()	Methods to retrieve individual request attributes

RWCString& GetESDT_Id() RWCString& GetRequestPriority() RWCString& GetRequestHomeDAAC() MsAcAddress& GetShipAddr() RWCString& GetReceiveDateTime() RWCString& GetStartDateTime() RWCString& GetFinishDateTime() RWCString& GetTimeOfLastUpdate() RWCString& GetShipDateTime() RWCString& GetFtpAddress() RWCString& GetFtpPassword() RWCString& GetDestinationNode() RWCString& GetDestinationDirectory() RWCString& GetRequestType() RWCString& GetProductId() RWCString& GetRequestOwnerId() RWCString& GetUpdatedById()	
SetRequest(const RWCString& orderId, RWCString& requestId, RWCString& parentId, MsAcUsrName& userName, RWCString& eMailAddr, RWCString& requestDesc, RWCString& requestStatus, RWCString& requestDistFormat, EcTInt& numFiles, EcTInt& numBytes, EcTInt& numGranule, RWCString& deviceId, RWCString& deviceDensity, RWCString& tapeFormat, RWCString& mediaType, RWCString& ESDT_Id, RWCString& requestPriority, RWCString& requestHomeDAAC, MsAcAddress& shipAddr, RWCString& receiveDateTime, RWCString& startDateTime, RWCString& finishDateTime, RWCString& timeOfLastUpdate, RWCString& shipDateTime, RWCString& FtpAddress, RWCString& FtpPassword, RWCString& destinationNode, RWCString& destinationDirectory, RWCString& requestType, RWCString& productId, RWCString& requestOwnerId, RWCString& updatedById)	Sets Request attributes
SetOrderId(const RWCString& orderIdIn) SetRequestId(const RWCString& requestIdIn) SetParentId(const RWCString& parentIdIn) SetUserName(const MsAcUsrName& usernameIn) SetEMailAddr(const RWCString& eMailaddrIn) SetRequestDesc(const RWCString& requestdescIn) SetRequestStatus(const RWCString& requeststatusIn) SetRequestDistFormat(const RWCString& requestdistformatIn)	Methods to set individual request attributes

SetNumFiles(const EcTInt& numfilesIn) SetNumBytes(const EcTInt& numbytesIn) SetDeviceId(const RWCString& deviceidIn) SetDeviceDensity(const RWCString& devicedensityIn) SetTapeFormat(const RWCString& tapeFormatIn) SetNumGranule(const EcTInt& numgranuleIn) SetMediaType(const RWCString& mediaTypeIn) SetESDT_Id(const RWCString& ESDT_IdIn) SetRequestPriority(const RWCString& requestpriorityIn) SetRequestHomeDAAC(const RWCString& requesthomedaacIn) SetShipAddr(const MsAcAddress& shipaddrIn) SetReceiveDateTime(const RWCString& receivedatetimeIn) SetStartDateTime(const RWCString& startdatetimeIn) SetFinishDateTime(const RWCString& finishdatetimeIn) SetTimeOfLastUpdate(const RWCString& TimeOfLastUpdateIn) SetShipDateTime(const RWCString& shipdatetimeIn) SetFtpAddress(const RWCString& FtpAddressIn) SetFtpPassword(const RWCString& FtpPasswordIn) EcTVoid SetDestinationNode(const RWCString& destinationNodeIn) SetDestinationDirectory(const RWCString& destinationDirectoryIn) SetRequestType(const RWCString& requestTypeIn) SetProductId(const RWCString& productIdIn) SetRequestOwnerId(const RWCString requestOwnerIdIn) SetUpdatedById(const RWCString& updatedByIdIn)	

Table 4.3-1. Class Name and Member Functions for Order and Request Tracking (3 of 3)

Class Name and Description	
EcAcOrderCMgr This class defines the client object to interface with the MSS Order and Request Tracking Server. The client object is used to create, update, retrieve and delete EcAcOrder objects and EcAcRequest object. The MSS Order and Request Tracking Server stores the objects in persistent storage at the MSS server. It also assigns unique order IDs and request IDs.	
Member Function Name	**Member Function Description**
EcAcOrderCMgr(const RWCString& ServerName)	Constructor: Creates the client object
EcUtStatus CreateOrder(EcAcOrder& order, RWCString& orderId, EcTInt& fundingStat);	Create an order by storing an order object in MSS's Order and Request tracking DB. Compares order price (in order object) to user's credit, fundingStat is set to INSUFFICIENT_FUNDS and no order is stored if user does not have sufficient credit. Otherwise, a unique order ID is stored and passed back in orderId. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus UpdateOrderStatus(const RWCString& orderId, const RWCString& status)	Update Order Status by orderId. Updates status field in DB for orderId with new value in status. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus UpdateOrder(EcAcOrder& order)	Updates the entire order DB record with the attributes passed in order. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus DeleteOrder(RWCString& orderId);	Deletes order specified by orderId from the database. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus RetrieveOrder(RWCString& orderId, EcAcOrder& order);	Retrieve Order by orderId. Retrieves order attributes of orderId from the database, builds and returns order object 'order'. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus RetrieveRequestOrder(RWCString& requestId, EcAcOrder& order)	Retrieve Order by requestId. Retrieves order attributes of the order for the requestId from the database, builds and returns order object 'order'. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus CreateRequest(EcAcRequest& request, RWCString& requestId);	Create an request by storing request in MSS's Order and Request tracking DB. Creates a unique request ID that is stored in DB and passed back in requestId. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus UpdateRequest(RWCString& requestId, RWCString& status)	Update Request Status by requestId. Updates status field in DB for requestId with new value in status. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus UpdateRequest(EcAcRequest& request)	Overloaded method to updates the entire request database record with the attributes passed in request. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK
EcUtStatus UpdateRequestStatus(RWCString& requestId, RWCString& status)	Update Request Status and automatic update associate order status. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus RetrieveRequest(RWCString& requestId , EcAcRequest& request);	Retrieve request by requestId Retrieves request attributes of requestId from the database, builds and returns request object 'request'. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus DeleteRequest(RWCString& requestId);	Deletes requestr specified by requestId from the database. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.

EcUtStatus RetrieveOrderByUsrName(RWCString& lastname, RWCString& firstname, EcAcOrderList& list)	Retrieve list of all orders for a user by lastName and firstName. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus RetrieveOrderList(RWCString& userId, long maxnumber, EcAcOrderList& list)	Retrieve list of all orders belonging to an userid. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus RetrieveRequestByUsrName(RWCString& lastname, RWCString& firstname, EcAcRequestList& list)	Retrieve list of all requests for a user by lastName and firstName. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.
EcUtStatus RetrieveRequestList(RWCString& orderId, long maxnumber, EcAcRequestList& list)	Retrieve list of all requests belonging to an orderId. EcUtStatus contains FAILED if unsuccessful otherwise it contains OK.

4.3.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user applications must be connected to OODCE (see section 3.5.1 DCE communications)
- user applications are written in C++
- MSS server is up and running with Sybase Request Tracking Data base defined.

4.3.4 End Conditions

The scenario terminates with the following end conditions:

- status of the order request is returned to the requesting application

4.3.5 Detailed Processing Steps

Figure 4.3-1 is an event trace of how applications would typically interface with MSS's Order and Request Tracking server. The Order and Request Tracking is accomplished in five main steps. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

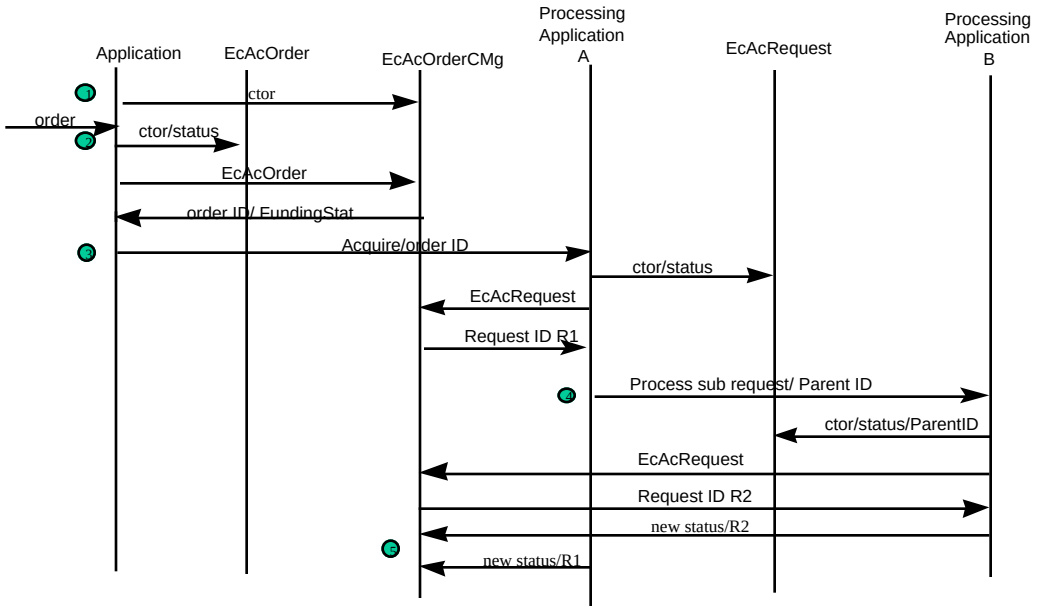

Figure 4.3-1. Order/Request Tracking Event Trace

Step (1) any application reporting status would create an instance of the Order Manager Client. The application would then wait for an order to process:

```
RWCString serverName( " cdsname" );
EcAcOrderCMgr  OrderManager( serverName );
```

Step (2) an order has been received and the application would create and instance of on order object, fill the order object with it's attributes (including status) and pass the order object to the order manager. The Order Manager checks for sufficient funds and returns the order ID:

```
EcAcOrder   order;

// set attributes
RWCString  userId("jdoe");
MsAcUserName  userName;

// fill userName object
RWCString  status( " NEW" );
EcTFloat price = 10.0;
•
•
// fill order object
order.SetUserId( userId );
order SetUserName( userName );
order.SetOrderStatus( status );
order.SetPrice( price );
•
•
```

```
// Send to Order Manager - Get funding status and order ID
EcTInt fundingflag;
RWCString orderId;
OrderManager. CreateOrder(order, orderId, fundingFlag);
if ( fundingFlag == INSUFFICIENT_FUNDS)
{
// don't process order
}
else
{
// continue processing
}
```

Step (3) the order is passed to processing application A. Application A creates an instance of a request object, fills the request object with its attributes (including status and order ID) and passes the request object to the Order Manager. The request ID of the object is returned.

```
// Process order request sent to Application A with  order information
EcAcRequest  request1;

// set attributes
RWCString  parentId;

// set null parent ID since this is first request for order
RWCString  statusA( " NEW" );
EcTint numGranule = 10;
•
•
// fill order object
request1.SetOrderId( orderId );
request1.SetUserName( userName );
request1.SetOrderStatus( statusA );
request1.SetParentId( parentId );
request1.SetNumGranule( numGranule );
•
•
// Send to Order Manager - Get Request ID for first request
RWCString requestIdR1;
OrderManager. CreateRequest(request1, requestIdR1);
```

Step (4) Application A sends a portion of the order to be process by Application B. Application B creates another instance of a request object, fills the request object with it's attributes (including status , order ID and parent request ID) and passes the request object to the Order Manager. The request ID of the second object is returned.

```
//  Process order request sent to Application B with  order information
    and Parent request ID:
//  requestIdR1
EcAcRequest  request2;


//  set attributes
RWCString  statusB( " Submitted" );
EcTint numGranule = 4;
•
•
```

```
// fill order object
request2.SetOrderId( orderId );
request2.SetUserName( userName );
request2.SetOrderStatus( statusB );
request2.SetParentId(requestIdR1);
request2.SetNumGranule( numGranule );
•
•
// Send to Order Manager - Get Request ID for first request
RWCString requestIdR2;
OrderManager. CreateRequest(request2, requestIdR2);
```

Step (5) Applications A and B update the status of their request:

```
// Application A
statusA = "Submitted";
OrderManager.UpdateRequest( requestIdR1, statusA );

// Application B
statusB = "InProduction";
OrderManager.UpdateRequest( requestIdR2, statusB );
```

4.4 Submitting Subscriptions and Receiving Notification

4.4.1 Introduction

Interfacing ECS Subsystem: Communications Subsystem (CSS).

The purpose of this service is to provide an interface to submit a subscription to the subscription server. EcClSubscription is the client side subscription which can be created using attribute information from advertisements.

Subscription to an event requires prior knowledge of EventID and the SubscriptionServerUR where the event resides. The subscriber obtains this information from the Advertisement Service. The subscriber can specify a set of constraints (qualifiers) in order to receive notifications for only the specific events which match the constraints. The user provides these constraints in the form of a Parameter = Value list (GlParameterList). The subscriber receives an E-mail notification when the event occurs.

The occurrence of an event prompts the Subscription Service to automatically send an E-mail notification to all subscriptions that have submitted a request for notification. The subscriber specifies the Notification Text to be delivered within the body of the E-mail. The subscriber's E-mail address is obtained by inquiring in the MsAcUserProfile database with the appropriate unique userID.

4.4.2 Classes / Member Functions Used

Table 4.4 -1 gives the classes and specific member functions used to **Submit a Subscription**.

Table 4.4-1. Class Name and Member Functions for Submit a Subscription

Class Name and Description	
EcClSubscription include file : "EcClSubscription.h" This class represents a single subscription on the client side. This class provides the data and behavior that is particular to the subscription on the client side.	
Submit()	Submit itself to the subscription server, and store this subscription persistently in the database.
SetStartDate(StartDate : RWDate)	Set date subscription becomes active
SetEndDate(ExpirationDate : RWDate)	Set expiration date for subscription
SetQualifiers(Qualifiers : GlParameterList)	Set the qualifiers used to filter subscribable events
setNotificationText(NotificationText : RWCString)	Set the text to be sent to user upon event occurence
Update()	Update this subscription with new information.
Cancel()	Cancel this subscription from the subscription database.

4.4.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user application must be connected to OODCE (see section 3.5.1 DCE Communications)
- user application must be written in C++ to send calls to the C++ objects
- Client has retrieved an advertisement object from the Advertising Server that represents the event that he/she is interested in.
- Client has retrieved the eventID and subscription server UR from the advertisement object.

4.4.4 End Conditions

The scenario terminates with the following end conditions:

- The subscription is submitted to the subscription server, and stored persistently in the database.

4.4.5 Detailed Process Steps

Submitting Subscriptions and Receiving Notification is accomplished in the eight main steps which are described below. The series of steps is also documented in the event trace in Figure 4.4-1. An explanation of how to use the event trace is given in the introduction to Section 4.

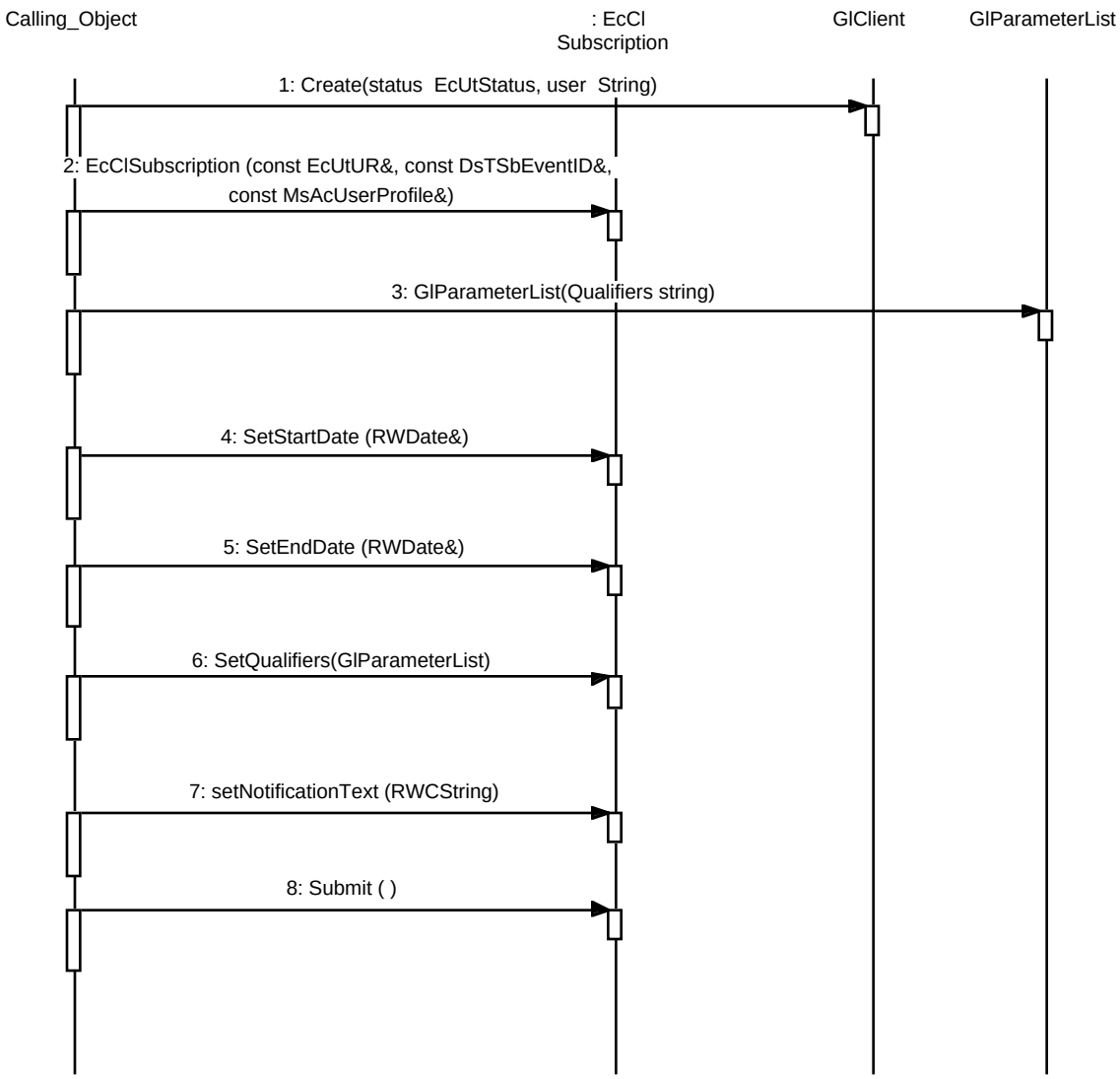

Figure 4.4-1. Submitting a Subscription Event Trace

Steps (1) and (2) an instance of the class EcClSubscription is constructed by the Calling Object with the subscription server UR and the event ID retrieved from the advertisement object. This step connects the client to the specified subscription server. Optionally, the default constructor, or the auto-filled constructor may be used for the same purpose.

Note: The creation of a user currently is performed using the GlClient object as shown in the code fragment. It is likely that the interface will be modified during Release B development to use the MsAcUserProfile object. Later versions of this document will specify the as-built interface.

```
#include <EcClSubcription.h>
EcUtStatus stat;
GlClient* user = GlClient::Create(stat, “user1”);
EcClSubscription* mySubscription=
                    new EcClSubscription(eventID, serverUR, user);
```

Step (3) subscription notification criteria is created. (This step may be omitted if the qualifiers has already been created, or event qualifiers is not applicable). An instance of the class GlParameter is constructed. The object accepts a parameter type (String) and a parameter value (Value). In this example, two parameters are created. An instance of class GlParameterList is created, and both parameters (GlParameter) are inserted into it.

```
GlStringP rangeEndDate(“1990/06/11”, “RangeEndingDate”);
GlStringP rangeStartDate(“1980/4/12”, “RangeStartDate”);
EndDate.SetDescription(“<=”);
StartDate.SetDescription(“>”);
GlParameterList myQualifiers(“Qualifiers”);
myQualifiers.insert(&rangeEndDate);
myQualifiers.insert(&rangeStartDate);
```

Steps (4) through (7), Set methods are used to set subscription attributes information: EventID, UserID, SubscriptionStartDate, SubscriptionExpirationDate, Qualifiers, and NotificationText. (This step may be omitted, if the auto-filled constructor is used).

```
RWDate myStartDate;    // today
RWDate myExpDate(“09/09/1999”);
RWCString notifText(“This is my notification text”);
stat = mySubscription->SetStartDate(myStartDate);
stat = mySubscription->SetEndDate(myExpDate);
stat = mySubscription->SetQualifiers(myQualifiers);
stat = mySubscription->SetNotificationText(notifText);
```

Step(8), finally the Submit() method of EcClSubscription is invoked by the Calling Object, passing the EventID, User, StartDate, ExpirationDate, Action and Qualifiers. This causes the subscription to be created in the server process space, and stored persistently in database.

```
stat = mySubscription->Submit();
```

4.5 Search for Advertisements

4.5.1 Introduction

Interfacing ECS Subsystem:

The purpose of this service is to allow user’s applications to search for advertisements about ECS products. The Advertising Service provides the interfaces needed to support application program defined interactive searching and retrieval of advertisements. Although there will be a single format for submitting advertisements to the service, advertisements are accessible via several different interfaces to support database and text searching, and retrieval according to several different viewing styles (e.g., plain ASCII text, interactive form, or HTML document).

A data server or other provider will advertise its data collections and services with the Advertising Service. The advertisement will include a listing of all products (and other Earth Science Data Types) available in the collection and a set of product attributes. Advertisements include directory level metadata, therefore, the attributes reflected in the advertising service include the ECS Core Metadata Directory-Level attributes that apply to collections. The client will send user queries which access only directory level metadata directly to the advertising service (rather than sending it as a distributed query to the various sites which provided the advertising information). A user who wishes to find out what data sets are available on the network can search (i.e., formulate a query) or browse (i.e., navigate through hyperlinked pages of advertisements) the advertising information. Both types of 'directory searching' are available on the user's desktop; the user can choose whichever approach is most convenient in the current work context.

4.5.2 Classes / Member Functions Used

Tables 4.5-1 give the classes and specific member functions used to **Search for Advertisements**.

Table 4.5-1. Class Name and Member Functions Search for Advertisements (1 of 2)

Class Name and Description	
IoAdApprovedAdvSearchCommand Parent Class: IoAdSearch This public class provides interfaces for applications to search the set of product advertisements by specifying options and criterion. The persistent data will be stored into a results list for additional searches or access. Users should set up options of how to search (filtering, patterns, how many results to return) and then call the search interfaces.	
Member Function Name	Member Function Description
SetAdvType(searchType:EcUtClassID)	This operation determines which advertisement type to search: Product, Service, or Provider.
SetTitle (matchTo: RWCString, matchType:MatchTypeEnum = Contain, logicType : LogicTypeEnum = ORType)	This operation searches for any Advertisement that contains a substring of matchTo in its Title and appends found advertisements to the current results set (if a matchType of Contain is selected). Alternatively, the search would look for a prefix of matchTo or an exact match of matchTo if the matchType was Prefix or Exact. The logicType parameter allows one to specify whether the advertisement is OR'ed or AND'ed with the other criteria that can be searched on.
SetGroup (group : const IoAdGroup&)	Sets which group to search on advertisements, for example, ECS, SCF, International Partners, etc.
GetResults (): IoAdAdvertisementList	This operation returns the matched Advertisement object to the user.

Table 4.5-1. Class Name and Member Functions Search for Advertisements (2 of 2)

Class Name and Description	
IoAdAdvertisementList Parent Class: RWTPtrSlist This class represents a list of products returned by the search for advertisements. This class behaves as a RWTPtrSlist with different copy semantics.	
Member Function Name	Member Function Description
Entries (): int	This operation returns the number of elements, in this case the number of products, in the list.
Operator[]: IoAdProduct	This operator selects one product from the list and return it to the caller.

Please note that only the member functions used in the scenario are presented. For a complete list of member functions see the 305-CD-022-002 document. Please also note that the member functions presented could be inherited from parent classes.

4.5.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user application must be connected to DCE (see section 3.5.1 DCE Communications)
- user application must be written in C++ to send calls to the C++ objects

4.5.4 End Conditions

The scenario terminates with the following end conditions:

- user application has found the advertisement for the specific product it was looking for.

4.5.5 Detailed Process Steps

The Search for Advertisements on the ADSRV is accomplished in four main steps, which are described below. The series of steps is also documented in the event trace in Figure 4.5-1. An explanation of how to use the event trace is given in the introduction to Section 4.

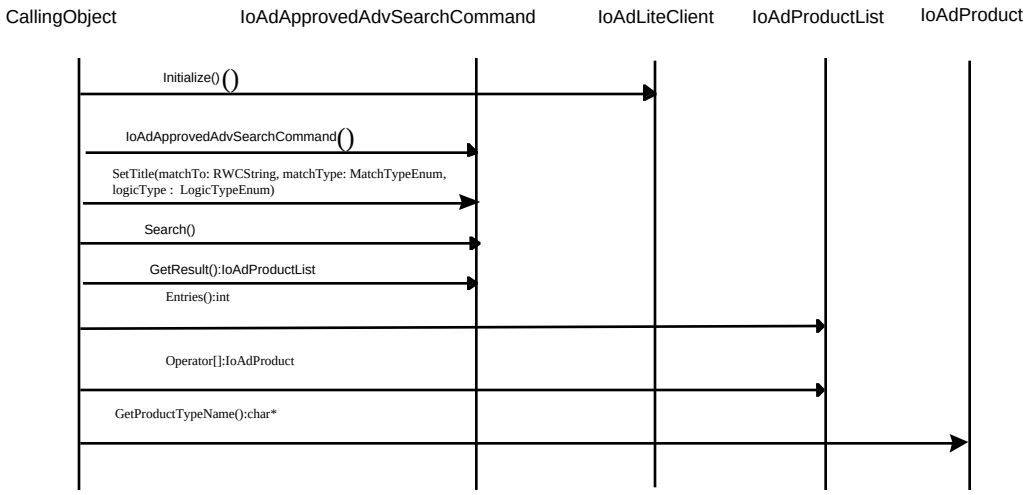

Figure 4.5-1. Search for Advertisements Event Trace

Step (1) the IoAdLiteClient object initializes the client-server communications by reading some environment variables at run-time. These environment variables will be documented in future versions of this document or in the design documentation for the Interoperability Subsystem (305-CD-022). The environment variables will specify which Advertising Service to connect to, thus the application program simply has to call the Initialize function of the IoAdLiteClient object.

```
IoAdLiteClient      client;
client.Initialize();
```

Step (2) the application program must specify the search constraints of the Advertising Service. In this interface we are not using ESQL, but setting some search constraints on the search command. The example shown in this step, searches for the word/acronym ASTER in the title of a product advertisement.

```
IoAdApprovedAdvSearchCommand     cmd();
cmd.SetTitle("ASTER", IoAdApprovedAdvSearchCommand::Contain,
             IoAdApprovedAdvSearchCommand::ANDType);
cmd.SetAdvType(IoAdProduct::GetOurIoAdAdvertisementClassID());
cmd.Search();
```

Step (3) the IoAdApprovedAdvSearchCommand then creates a IoAdApprovedAdvList object to append found advertisements to the current results set. The application program calls the GetResults method of the IoAdApprovedAdvSearchCommand object to return the matched Product advertisements.

```
IoAdApprovedAdvList ad = cmd.GetResults();
```

Step (4) the calling object then calls the IoAdApprovedAdvList object to iterate through the products.

```
int  numOfFoundAd;         // number of Ad found in the search
int  foundFlag = FALSE;    // flag to signal found searched Ad
numOfFoundAd = ad.Entries();
const IoAdApprovedAdvList related;
for (i = 0; i < numOfFoundAd && foundFlag==FALSE; i++ )
{
       // print out the name of the product.
       cout << “Product Name = “ << ad[i].GetAdv()->GetTitle;

       // print or process the other attributes of the
       // product, such as find the service URs.
       related = ad.GetRelatedAdvs();
       for (j = 0; j < related.entries(); j++)
       {
              if (
              IoAdAdvertisement::KindOfAdv(related[j].GetObjectID()
                     ==  IoAdService::GetOurAdvertisementClassID())
              {
                     // allocate a service advertisement and
                     // read the data.
                     IoAdService serv = (IoAdService) related[j];
                     // process data.
              }
}
```

4.6 Searching Data Dictionary

4.6.1 Introduction

Interfacing ECS Subsystem: Data Dictionary (DDICT) component in the Data Management Subsystem (DMS)

The purpose of this service is to provide access to databases containing information about data objects, their attributes, their operations, and the domains of the attributes. The DDICT describes the data objects accessible through data servers, LIMs, DIMs, and GTWAYs. The DDICT is used for informational support by users to retrieve definitions of the available items and as infrastructural support to the other CSCIs within the Data Management Subsystem (LIMGR, DIMGR, and GTWAY). Clients that search ECS holdings should search the DDICT to determine the proper names of attributes and collections in order to construct the correct ESQL query.

4.6.2 Classes / Member Functions Used

The following table gives the classes and specific member functions used to **Search the Data Dictionary**.

Table 4.6-1. Class Name and Member Functions for Search the Data Dictionary (1 of 2)

Class Name and Description	
DmDdClRequestServer include file : "DmDdClRequestServer.h" This class is inheriting from EcCsRequestServer_C. It is used to manage the user session and to create objects capable of communicating asynchronously between a client and a server.	
Member Function Name	Member Function Description
DmDdClRequestServer(server :EcUrUR, user :MSSUserProfile &)	Constructor called by a client application. User's profile is passed as a parameter.
NewSearchRequest(request :DmDdClrequest *)	This method is called by the client application to a new search request. It returns a pointer to an asynchronous object DmDdClSearchRequest, which will be used to actually submit the search request.

Table 4.6-1. Class Name and Member Functions for Search the Data Dictionary (2 of 2)

Class Name and Description	
DmDdClRequest include file : "DmDdClRequest.h" This class will handle a specific request to the DDICT servers. Once a request is issued and fully satisfied, this object can be reused to initiate other requests of the same type to the server. Each request is handled asynchronously, with status of the request being returned to the calling object through a callback function. This class inherits from EcCsAsynchRequest_C from the SRF Key Mechanism.	
Member Function Name	Member Function Description
DmDdClRequest ()	Constructor called by a client application.
SetCallBack(DmDdCallBack *) : void	Allows the application program that created the request to be notified when a state change happens within the request. A state change for example will occur when the request has completed its query and the results are available. The function supplied has to accept two parameters, myRequest : DmDdClRequest * and myState, an enumerated type inherited from SRF. myRequest will be used by the caller to identify which request is calling back so that a single callback could potentially be used for multiple requests.
SetQuery(query : RWCString) : EcUtStatus	This method will accept RWCString as the search constraint and pass the argument to the server. This just sets the constraint in the client side object. The request is not submitted to the server until the Submit method is called. It returns a status.
Submit() : EcUtStatus	When the application program invokes Submit, the request will encapsulate all commands or constraints into a message object and ship that message object to the server side where it will be processed. The message object is determined by the request type and it is shipped using the EcCsMsgHandler object (SRF).
GetResults(startpoint :int, endpoint :int) : GlParameterList	This method will allow the application program to retrieve search results . Specified are the startpoint and the endpoint which allow for a range of results to be returned. This allows callers to customize the size of the results to the application program's hardware configuration.

4.6.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- user application must be connected to DCE (see section 3.5.1 DCE Communications)
- user application must be written in C++ to send calls to the C++ objects
- user application must use UR mechanism to point to correct DDICT. The UR can be hard coded or retrieved from the Advertising Service prior to starting this scenario.
- user application must have instantiated an MSS User Profile object to identify the user using this service. If the user profile is passed as null, the system will assume guest privileges.

4.6.4 End Conditions

The scenario terminates with the following end conditions:

- user application called back by DDICT when operation complete
- DDICT DmDdClRequest object returns results to client program.

4.6.5 Detailed Process Steps

The Searching of the Data Dictionary is accomplished in seven main steps, which are described below. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

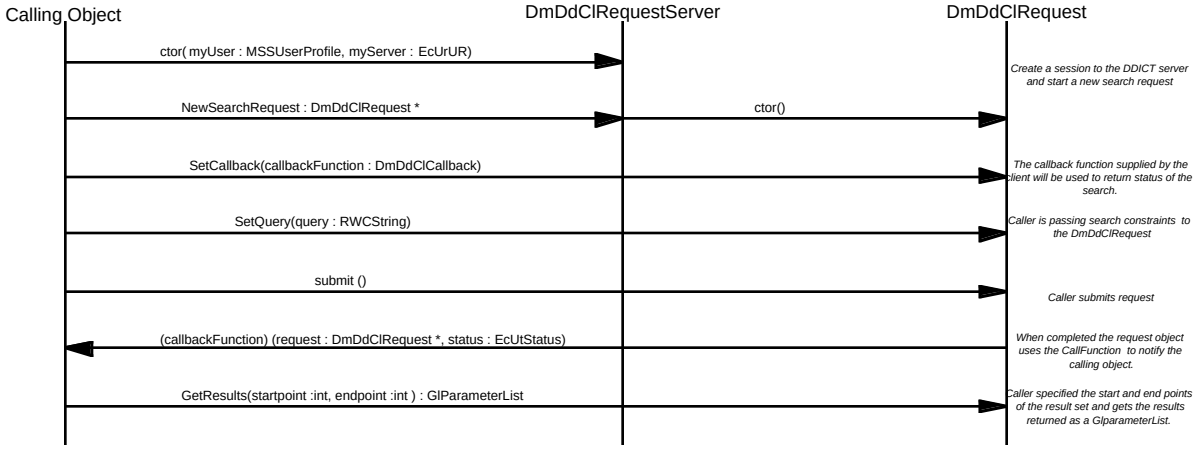

Figure 4.6-1. Searching Data Dictionary Event Trace

Step (1) the calling object initiates a session with the DDICT by creating a DmDdClRequestServer object. The DmDdClRequestServer object is a synchronous session object which establishes a connection to a server and instantiates a server-side request factory object. The server connection is established to the server with the UR specified in the constructor.

```
// retrieve server UR from advertising
EcURUR       serverUR = ad.GetProviderObject();
DmDdClRequestServer server(user, serverUR);
```

Step (2) the calling object initiates the creation of a new request by using the method NewSearchRequest from the DmDdClRequestServer object. The EcCsMsgHandler object of the Server Request Framework (SRF) creates a DmDdClSearchRequest object upon notification that the correlated server side object has been created. The DmDdClSearchRequest object is an asynchronous object that inherits from the SRF.

```
DmDdClRequest request;
EcUtStatus    status;
status = server.NewSearchRequest(&request);
```

Step (3) the calling object populates the DmDdClRequest with the search constraints. The query specified is an example. It would extract all attributes from the DDICT who belong to the collection ‘AST03’. See the SDPS Database Design and Database Schema Specification for the ECS Project (311-CD-008-001) for more details about the DDICT schema and the meanings of the attributes.

```
RWCString    query = “select AttributeName from DDICT where
ShortName =‘AST03’”;
request.SetQuery(query);
```

Step (4) the application program specifies a callback function to the request object so that the calling object can be notified of the completion of the request. The callback function is shown in Step 7 where we actually get the results.

```
DmDdCallback myCallback (DmDdClRequest *request, EcCsState state)
{
       //Process the request
}
request.SetCallBack(*(myCallback));
EcUtStatus   status;
status = request.Submit();
```

Step (6) the calling object's callback is invoked upon each state change. The application program can choose which states to monitor progress on.

```
// DDICT library transparently calls myCallback on every state change.
```

Step (7) the calling object then specifies the range of results it can accommodate and retrieves the results specified from the DmDdClRequest object. For example, the client can specify that it wants results 1 through 100 returned to reduce the amount of data being returned to the user. The return type is a GlParameterList that contains the attributes as requested in the query. A GlParameterList inherits from a RogueWave ordered vector template.

```
DmDdCallback myCallback (DmDdClRequest *request, EcCsState state)
{
       if (state == COMPLETE)
       {
              GlParameterList     list;
              list = request->GetResults(1, 100);
              // process the results list.
              For (i = 0; i < list.entries(); i++)
              {
                     // Get the AttributeName and process it.
              }
       }
```

4.7 Submitting Advertisements

TBD-14

4.8 Submitting Automatic Ingest Request

4.8.1 Introduction

Interfacing ECS Subsystem: Automatic Network component in the INGEST Subsystem (INS)

The purpose of this service is to provide an interface for OODCE clients to submit a Delivery Availability Notice (DAN) to the Ingest subsystem. The OODCE Client sends a DAN message to ECS, specifying the names of the data files, file sizes, file dates and times, number of files, and file locations for the files available for ECS to archive Figure 4.8-1 shows the contents of a sample DAN.

ECS verifies the user is authorized and validates the DAN, and sends the corresponding handshake control message, the Data Availability Acknowledgment (DAA), which supports the disposition of the DAN. Each DAN is distinguished from the others by the sequence number and processor identifier which created it. These parameters are included in the DAN message. After all DANs have been acknowledged, the OODCE Client closes the session by sending ECS a Close Session message, and terminates the connection.

When ready, ECS begins the kftp file transfer process and transfers all of the files in each error-free file group listed in the DAN. Each file is verified by checking its name against DAN information; metadata is extracted; and the file transfer result is logged in the Data Delivery Notice (DDN). After all the files have been transferred, ingested and archived, or when all attempts have been exhausted, ECS sends the OODCE Client a DDN to notify whether the files were successfully archived and/or identify errors associated with individual files for a particular DAN. Only complete file groups that are transferred without error are ingested and archived. The OODCE Client responds with the corresponding handshake control message, the DDA. A DDA of 2 indicates that the OODCE Client had problems processing the DDN through their system (i.e., database failure). ECS will resend the DDN at a later time. The resend time is tunable parameter.

An alternative to the Automated Network Ingest is the Ingest Polling or HTML interface. The Ingest HTML interface allows the data provider to submit Ingest request, to view Data Delivery notice and to monitor status of data provider on-going Ingest request. Ingest Polling provides two interfaces, Polling with Product Delivery Record (PDR) and Polling without PDR. In the Polling Ingest with PDR interface, ECS periodically checks an agreed-upon network location for a PDR file. If a PDR file is located ECS gets data from the source, specified by the PDR file, within a system-tunable time window. In the Polling Ingest without PDR file interface, ECS periodically checks an agreed-upon network location for available data. All data in the location is assumed to make up a collection of ingest data with one file per data granule. If data is located ECS gets data from the source within a system-tunable time window.

The following pages give a detailed explanation of the Automated Network Ingest interface only. Future versions of this document will provide a detailed explanation of the Polling with PDR interface and the Polling without PDR interface.

```
ORIGINATING_SYSTEM = OODCE Client
CONSUMER_SYSTEM = ECS_GSFC_1;
DAN_SEQ_NO = 5326;
TOTAL_FILE_COUNT = 3;
AGGREGATE_LENGTH = 649678;
EXPIRATION_TIME = 1998-11-12T20:00:00Z;
OBJECT = FILE_GROUP;
        DATA_TYPE = 1A11;
        DATA_VERSION = 1;
        NODE_NAME = tsdssrv1.gsfc.nasa.gov;
        OBJECT = DP_CIO;
                DIRECTORY_ID = /oodce client/tmi/1a;
                FILE_ID = <tsdis file name>;
                FILE_TYPE = METADATA;
                FILE_SIZE = 1100;
        END_OBJECT = DP_CIO;
        OBJECT = FILE_SPEC;
        DIRECTORY_ID = /oodce client/tmi/1a;
                FILE_ID = <tsdis file name>;
                FILE_TYPE = SCIENCE;
                FILE_SIZE = 242120;
                BEGINNING_DATE/TIME = 1998-11-08T18:36:18Z;
                ENDING_DATE/TIME = 1998-11-08T20:10:07Z;
        END_OBJECT = FILE_SPEC;
END_OBJECT = FILE_GROUP;
OBJECT = FILE_GROUP;
        DATA_TYPE = 1B11BR;
        DATA_VERSION = 1;
        NODE_NAME = tsdssrv1.gsfc.nasa.gov;
        OBJECT = FILE_SPEC;
                DIRECTORY_ID = /oodce client/tmi/1b;
                FILE_ID = <oodce client file name>;
                FILE_TYPE = BROWSE;
                FILE_SIZE = 242120;
                BEGINNING_DATE/TIME = 1998-11-08T18:36:18Z;
                ENDING_DATE/TIME = 1998-11-08T20:10:07Z;
        END_OBJECT = FILE_SPEC;
END_OBJECT = FILE_GROUP;
```

Figure 4.8-1. Sample DAN PVL (OODCE Client)

4.8.2 Classes / Methods Used

Table 4.8-1 gives the classes and specific member functions used to **Submit Automatic Ingest Request**.

Table 4.8-1. Class Name and Member Functions for Submit Automatic Ingest Request

Class Name and Description	
InAutoIngestIF_1_0 include files : "InAutoNtwkIngestC.H" , "PktStructC.H", "InAuCreateSessIFC.H" This public, distributed class and include files are generated by compiling the Interface Definition Language (IDL) files InAutoNtwkIngestIF.idl, PktStruct.idl and InCreateSessIF.idl. .	
Member Function Name	**Member Function Description**
DANMessage(DANLength, DRBuffer, DAABuffer, ErrorStatus)	Submits a DAN to INGEST subsystem.

4.8.3 Start Conditions / Initialization

The scenario assumes the following starting conditions:

- The data provider application must be written in C++ to send calls to the C++ objects
- The data provider application must use an UR mechanism to point to correct Automatic Network Ingest Server.
- The interface to the Automated Network Ingest is OODCE-based. If the data provider is not a DCE client, a gateway exists between the data provider and the IMS/DADS to convert RPCs to TCP/IP socket services and vice versa. If the data provider is a DCE client, the data provider must be registered in the ECS ACL Database and have access to ECS DCE cells.
- The data provider must have a staging area in the ECS user push staging area.

4.8.4 End Conditions

The scenario terminates with the following end conditions:

- If the DAN message fails validation, the scenario terminates when the provider receives a DAA containing the error status.
- If the DAN message passes validation, the DAA returned indicates success. The scenario terminates when the data provider receives a DDN indicating success/failure status of the Ingest request and returns a DDA to ECS indicating the receipt of the DDN.

4.8.5 Detailed Process Steps

Submitting an Automated Network Ingest Request is accomplished in eight main steps. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

TBD-16

Figure 4.8-2. Submitting Automated Network Ingest Request

Step (1) the OODCE client creates an instance of the class InAutoIngestIF_1_0.

```
    DCENsiobject serverDCEobject = new DCENsiobject(ServerECSDCECellEntry);
InAutoIngestIF_1_0  automatedIngest(serverDCEobject );
```

Note: There are other constructors that the user application can instantiate. Please peruse the IDL generated file InAutoNtwkIngestIFC.H to see which constructor is appropriate for your application.

Step (2) the client specifies its Authentication information. The Automated Network uses packet based DCE authentication. The clientPrincipleName parameter is the ACL database entry associated with the client. The other parameters are DCE constant that are defined in the DCE header files

```
automatedIngest.SetAuthInfo(clientPrinciplelName,  // char *
rpc_c_protect_level_pkt_integ,
rpc_c_authn_dce_secret,
(rpc_auth_identity_handle_t) NULL,
rpc_c_authz_dce);
```

Step (3) the client sends the DAN message to the server by calling the DANmessage method of the InAutoIngestIF_1_0 class. The DANMessage RPC returns a DAA message and status upon completion.

```
automatedIngest.DANMessage(DANmessageLength, // long int [input]
DANmessagePtr       // unsigned char**  [input]
DAAMessagePtr,      //PktStruct **  [output]
DAAStatus);         //error_status_t *  [output]
```

Step (4) The Server receives the DANMessage RPC and then authenticates the client. The server validates the DR message. If the DAN message is valid, then the DAN is staged in the ECS user area and the request is inserted into the Ingest database.

Step (5) A DAA message and error/success status are returned to the client

Step (6) ECS ingests the data

Step (7) Once the Ingest request is processed, a Data Delivery Notice (DDN) is sent to the OODCE client via OODCE Remote Procedure Call. The DDN provides completion status for ingest request. The DDN is sent by instantiating the CsGWTranferPkt_1_0 (IDL generate client stub) class and calling the TransferPacket method.

```
CsGWTransferPkt_1_0  Client(ClientECSDCECellEntry);
Client.TransferPacket(DDNMessageLength, //  long int  [input]
DDNMessageBuffer,       //  unsigned char*  [input]
TheReplyPacket,         //  PktStruct** [output]
DDNStatus);             //  error_status_t *  [output]
```

The DDNMessageLength and DDNMessageBuffer are input parameters. TheReplyPacket and DDNStatus are output parameters. TheReplyPacket is the Data Delivery Acknowledgment (DDA) which indicates that the OODCE client had problem processing the DDN through their system

Step (8) The OODCE client receives the TransferPacket RPC, and sends a DDA and the appropriate status to the ECS Ingest server. TransferPacket is a server stub on the OODCE client side of the interface.

```
CsGWTranferPkt_1_0_Mgr::TransferPacket(long int DDNMessageLength,
unsigned char* DDNMessageBuffer,
PktStruct** TheReplyPacket,
error_status_t*  DDNStatus)
{
char* retmsg = " I received your message";
int retmsglen =  strlen(retmsg) + 1;
       *TheReplyPacket =
`             (PktStruct *)rpc_ss_allocate(retmsglen + sizeof(PktStruct));
if (*TheReplyPacket == NULL )
{
             *ErrorStatus = !error_status_ok;
       return ;
}
       (*TheReplyPacket)->theLength = retmsglen;
memcpy( (*ThReplyPacket)->theMsg, retmsg, retmsglen -1);
*ErrorStatus = error_status_ok;
return;
   }
```

4.9 Update Metadata

4.9.1 Introduction

Interfacing ECS Subsystem: SDSRV
The purpose of this service is to provide the user the ability to update the attributes of an existing metadata object associated with data that ECS has ingested from that user.

4.9.2 Classes / Member Functions Used

Table 4.9-2 gives the classes and specific member functions used to **Update Metadata**.

Table 4.9-1. Class Name and Member Functions Update Metadata (1 of 6)

Class Name and Description	
DsClESDTReferenceCollector include file : "DsClESDTReferenceCollector.h" This public, distributed class is a specialization of the Collector class which handles DsClESDTReferences. This class provides the normal operations for ESDTReferences, the ability to handle requests, working-collection synchronization, and sessions. It also contains private operations to hand the ESDTReference-level actions to the data server	
Member Function Name	Member Function Description
DsClESDTReferenceCollector (server :DsShESDTUR &, client :GlClient &)	The constructor expects the Universal Reference (UR) of the server and the client to which it will connect.

Table 4.9-1. Class Name and Member Functions Update Metadata (2 of 6)

Class Name and Description	
GlStringP include file : "GlStringP.h", "GlParameter.h", "GlAll.h" This public class allows the capture of the command list or the results list	
Member Function Name	Member Function Description
GlStringP (value : RWCString, name : RWCString)	The constructor creates an GlStringP object taking the name and value as specified

Table 4.9-1. Class Name and Member Functions Update Metadata (3 of 6)

Class Name and Description	
GlParameterList include file : "GlParameterList.h" This public class allows the capture of the command list or the results list	
Member Function Name	Member Function Description
GlParameterList ()	The constructor creates an empty GlParameterList object
at (i)	The at() method allows access to the individual GlParameters in a GlParameterList at the ith entry on the list.
insert (parameter : GlParameter &)	The insert() method allows insertion of the individual GlParameters into a GlParameterList at the next entry in the list.

Table 4.9-1. Class Name and Member Functions Update Metadata (4 of 6)

Class Name and Description	
DsClCommand include file : "DsClCommand.h" This public, class is a specialization of the DsCommand for client interfaces. Adds constructors that ease the building of commands based on advertisements, or special direct commands that are "built-in" to the data server and do not correspond to advertisements. The commands are constructed by use of the GlParameterList Class	
Member Function Name	Member Function Description
DsClCommand (service :RWCString &, ParamList :GlParameterList, commandCatagory :DsEShSciCommandCatagory &)	The constructor expects the service name the parameters which need to be updated and the command category

Table 4.9-1. Class Name and Member Functions Update Metadata (5 of 6)

Class Name and Description	
GlCallback include file : "GlCallback.h" This public class allows the status of the request to be made available to the client	
Member Function Name	Member Function Description
GlCallback ()	The constructor creates an empty GlCallback object

Table 4.9-1. Class Name and Member Functions Update Metadata (6 of 6)

Class Name and Description	
DsClRequest include file : "DsClRequest.h" This public class is a specialization of the DsRequest for client interfaces. Allows the client to compose a request and submit it to the data server. Once submitted, the status may be polled, or a callback can be provided that is triggered on every status change.	
Member Function Name	Member Function Description
DsClRequest (command :DsClCommand &, priority :DsEShSciPriority &)	The constructor expects the commands constructed by use of the GlParameterList Class and the optional command priority (Default is NORMAL).
Submit(collector/dataServer :DsClESDTCollector *, domain :DsTShRequestDomain &)	Used to submit a request to be executed by a single, specific ESDT. The request is in turn submitted to the "implied" DsClESDTReferenceCollector. i.e. the one the DsClESDTReferenceCollector holds a pointer to. Optionally the request may be submitted with a domain. (Default is rwnil)

4.9.3 Start Conditions/Initialization

The scenario assumes the following starting conditions:

- user application must be written in C++ to send calls to other C++ objects

4.9.4 End Conditions

The scenario terminates with the following end conditions:

- user application called back by Science Data Server when operation complete

4.9.5 Detailed Process Steps

The updating of metadata within the Science Data Server is accomplished in the following 6 steps, which are described below. The series of steps is also documented in the event trace below. An explanation of how to use the event trace is given in the introduction to Section 4.

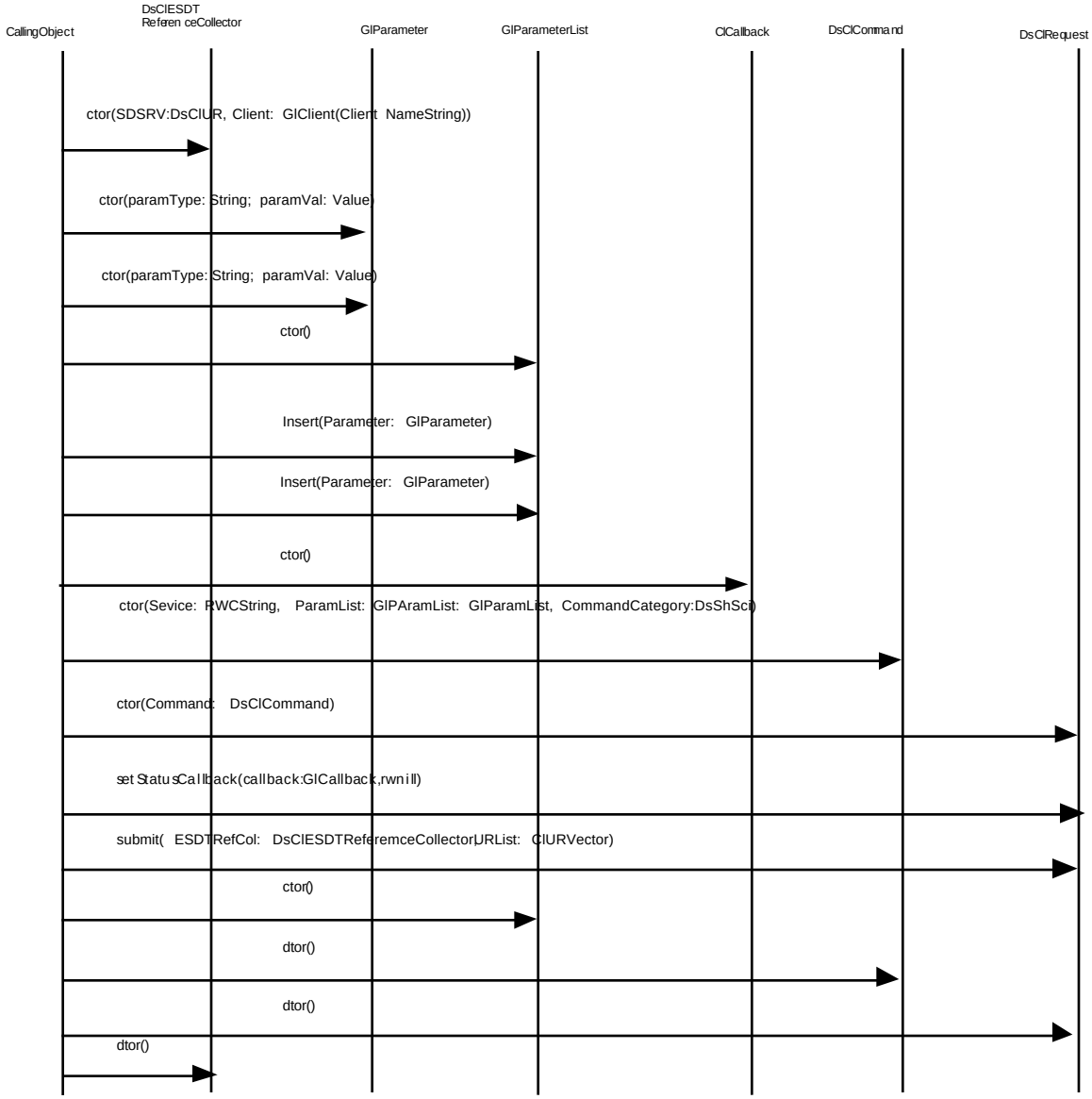

Figure 4.9-1. Update Metadata using the SDSRV event trace

Step(1) an instance of the class DsClESDTReferenceCollector is constructed by the Calling Object. This step is represented by the first(top) event in the event trace.

```
GlClient    theClient("AnyUser");
GlUR        theServer("TheHostName");
DsClESDTReferenceCollector ESDTRefCol(theServer, the client);
```

Step(2) the calling object needs to construct a list of attributes which are to be updated. This are individually represented by GlParameter classes of the correct Gl type to carry the data and are inserted into a GlParameterList class. Two GlStringPs are to be updated in the example.

```
GlStringP* myAttribute1 = new GlStringP("processed
once","ReprocessingActual");
GlStringP* myAttribute2 = new GlStringP("no further update
anticipated","ReprocessingPlanned");

// Create a list
GlParameterList* myAttList = new GlParameterList;

// add attributes
myAttList->insert(*myAttribute1);
myAttList->insert(*myAttribute2);
```

Step(3) an instance of the GlCallback object is constructed.

```
GlCallback myCallback;
```

Step(4) the calling object constructs an update command by creating an instance of DsClCommand. The DsClCommand object receives a service type, a GlParameterList containing the attributes to be updated in this case and a Command Category, which is WC (WorkingCollection) by default.

```
// Create Update command object
DsClCommand* myCommand = new DsClCommand("updateMetadata",myAttList,
DsShGlobal::WC);
```

Step(5) the calling object constructs a request by creating an instance of the DsClRequestcommand

```
// Make update commmand
DsClRequest myRequest(*myCommand);
```

Step(6) the request is submitted to the server by using the Submit method

```
// now submit this request to the sdsrv & wait for a return
myRequest.Submit(ESDTRefCol);
```

Step(7) the instance of the DsClReferenceCollector invokes the instance of GlCallback, which notifies the Calling Objects that the search is complete. This step is represented by the next event in the Event Trace.

Step(8) Need to assess the results. The Science Data Server returns a parent result pointer from the request. Create three instances of GlParameterLists more needed for multiple commands. Only one command was issued in this example.

```
const GlParameterList &requestResults = myRequest.GetResults() ;
```

Step(8) Inspect the request results for each command issue. Only one results list is expected in response to one command.

```
commandResults = (GlParameterList*)requestResults.at(0);
```

Step(9) Call DsClCommand destructor and recursively delete the GlParameters from the GlParameterLists followed by a call to the desctructor. Needs to be done for all newed GlParameterLists.

```
DsClCommand::~DsClCommand();
myAttList->RecursiveDelete();
delete myAttList;__
```

5. API Object Descriptions

5.1 General

The API object descriptions are found in Chapter 5 of DID 313 Release B CSMS/SDPS Internal Interface Control Document.

This page intentionally left blank.

Appendix A. Work-off Plan

IDD Issue #	IDD Para. #	Issue Priority*	IDD Issue Type - Description	Work-off Plan Task(s)	Projected Resolution Date
1	3.5	C	**TBS** - description of ECS key mechanisms.	Work is in progress. When the text to describe the key mechanisms has been completed, it will be added.	Competed
2	3.5.1	B	**TBS -** description of DCE requirements	Work is in progress. When the text to describe the DCE requirements has been completed, it will be added.	Completed
3	3.5.2	B	**TBS** - description of security requirements	Work is in progress. When the text to describe the security requirements has been completed, it will be added.	Completed
4	3.6	C	**TBS -** description of required software libraries	Work is in progress. When the text to describe the required software libraries has been completed, it will be added.	Completed
5	3.7	A	**TBS -** description of integration and testing process	Work is in progress. When the text to describe the I&T process has been completed, it will be added.	Completed
6	3.8	A	**TBS** - description of process to gain approval for use of API	The process does not presently exist. Will research the process and add data as part of maturation process of the system.	Completed
7	App. C	A	**TBS -** supply philosophy, hints, and example	Work is in progress. Gathering philosophy information and hints. Will provide example	6/1/97
8	3.5.1	C	**TBD -** provide required version of DCE	Contacted appropriate group to receive information. Will provide when available.	Completed
9	3.5.1	C	**TBD -** Describe how user registers his DCE cell	Researching question and will provide text when available.	Completed
10	3.6	C	**TBD -** provide required version of OODCE	Researching question and will provide text when available.	Completed
11	4.1	A	**TBD -** complete code fragments for Search for Data service	Will add to document when updated code is available	4/1/97
12	4.2.1	B	**TBD -** add code fragment for ftp'ing data across	Will add to document when updated code is available	4/1/97

IDD Issue #	IDD Para. #	Issue Priority*	IDD Issue Type - Description	Work-off Plan Task(s)	Projected Resolution Date
13	4.2.2	A	**TBD -** provide section describing how to order and receive data via the data server	Will add to document when information is available	Completed
14	4.7	A	**TBD -** provide section describing how to submit advertisements	Will add to document when information is available	4/1/97
15	4.9	A	**TBD -** provide section describing how to update metadata	Will add to document when information is available	Completed
16	4.8.2	B	**TBD -** provide event trace	Will add to document when information is available	4/1/97
17	3.8	C	**TBD** - provide URL for updated COTS information	Will Add before as comment at the CCB.	10/30/96
18	4.2.2	B	**TBD** - add code fragment in sec. 4.2.2	Will add when updated code becomes available	4/1/97

* Issue Priority Definition:

A = Design impact. E.g., unresolved interface.

B = Minimal design impact. E.g., content or format of a specific field unresolved.

C = No design impact - administrative detail. E.g., reference document # not available.

Appendix B. ECS Philosophy and Tips

TBS-7

This page intentionally left blank.

Abbreviations and Acronyms

ACL	Access Control List
ADSRV	Advertising Service
API	Application Programming Interface
CCB	Configuration Control Board
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CLS	Client Subsystem
CSCI	Computer Software Configuration Item
CSMS	Communications and System Management Segment
DAA	Data Availability Acknowledgment
DAAC	Distributed Active Archive Center
DADS	Data Archive and Distribution System
DAN	Data Availability Notice
DBMS	Database Management System
DCE	Distributed Computing Environment
DCN	Document Change Notice
DDA	Data Delivery Acknowledgment
DDICT	Data Dictionary
DDN	Data Delivery Notice
DFS	Distributed File Service
DID	Data Item Description
DIM	Distributed Information Manager
DIMGR	Distributed Information Manager CSCI
DLL	Dynamic Linked Library
DMS	Data Management Subsystem
DPS	Data Processing Subsystem

DSS	Data Server Subsystem
ECS	EOSDIS Core System
EDOS	EOS Data Operations Systems
EMC	Enterprise Monitoring and Coordination
EOC	EOS Operations Center
EOS	Earth Observing System
EOSDIS	EOS Data and Information System
ESDIS	Earth Science Data and Information System
ESDT	Earth Science Data Type
ESQL	Earth Science Query Language
FOS	Flight Operations Segment
GSFC	Goddard Space Flight Center
GTWAY	Version 0 Gateway
GUI	Graphical user interface
HTML	HyperText Markup Language
IDD	Interface Definition Document
ICD	Interface Control Document
IMS	Information Management System
INS	Ingest Subsystem
IOS	Interoperability Subsystem
ISS	Internetworking Subsystem
LAN	Local Area Network
LIM	Local Information Manager
LIMGR	Local Information Manager CSCI
LSM	Local System Management
NASA	National Aeronautics and Space Administration
OOD	Object Oriented Design
OO-DCE	Object Oriented Distributed Computing Environment
OSF	Open Software Foundation

OSI-RM	Open Systems Interconnection Reference Model
MSS	Management SubSystem
NSI	NASA Science Internet
PLS	Planning Subsystem
PSCN	Program Support Communications Network
rpc	remote procedure call
RW	Rogue Wave - commercial libraries
SCF	Science Computing Facility
SDPS	Science Data Processing Segment
SDSRV	Science Data Server
SMC	System Monitoring and Coordination Center
TBR	To Be Reviewed
TBD	To Be Determined
TBS	To Be Supplied
UR	Universal Reference
URL	Universal Reference Locator
V0	Version 0
WAN	Wide Area Network
WWW	World Wide Web

This page intentionally left blank.